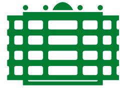

Fakultät für Informatik

Professur für Rechnerarchitektur

Diploma thesis

Developing a VIA-RPI for LAM

Ralph Engler, Tobias Wenzel

Chemnitz, 30th January 2004

Prüfer: Prof. Dr.-Ing. Wolfgang Rehm

Betreuer: Dipl.-Inf. Mario Trams

Abstract

LAM (Local Area Multicomputer) is an MPI programming environment and development system for heterogeneous computers on a network. With LAM, a dedicated cluster or an existing network computing infrastructure can act as one parallel computer to solve one problem. LAM includes different modules called RPI (Request Progression Interface) to support different kinds of communication (i.e. TCP/IP, shared memory, GM). VIA (Virtual Interface Architecture) is a communication principle with the goal to improve performance of distributed applications by reducing the latency associated with critical message passing operations. The goal of this diploma thesis is to develop a RPI module for the LAM/MPI software that uses this communication principle. This work is a cooperation of Ralph Engler and Tobias Wenzel.

Contents

List of Figures iv

1 Introduction 1

1.1 Current state of related works . . . 1

1.2 Goals of this work . . . 2

2 LAM-MPI - a high quality implementation of the MPI Standard 3

2.1 Features of LAM-MPI . . . 4

2.2 The Architecture of LAM-MPI . . . 6

2.3 SSI - The System Services Interface . . . 7

3 Virtual Interface Architecture 10

3.1 Design . . . 10

3.2 Basic concepts . . . 13

3.3 Communication principles . . . 14

3.4 Notification . . . 14

3.5 M-VIA -A high performance modular VIA for Linux . . . 15

4 LAM-RPI for M-VIA 17

4.1 Overview . . . 17

4.2 The interface to the LAM framework . . . 18

4.2.1 RPI API functions . . . 18

4.2.2 Basic data structures 20
4.3 Transmission principle 20
4.3.1 Send/Receive vs. RDMA-Write/Read 20
4.3.2 Immediate data 22
4.3.3 Empty receive queue problem 23
4.4 Message classification 24
4.4.1 Tiny protocol 26
4.4.2 Short protocol 27
4.4.3 Long protocol 28
4.5 A requests life 29
4.5.1 Building 29
4.5.2 Starting 29
4.5.3 Processing 30
4.5.4 Destroying 31
4.6 Memory management 32
4.6.1 Incoming envelopes plus tiny and short messages 32
4.6.2 Outgoing envelopes and tiny messages 33
4.6.3 Long and outgoing short messages 35
The libdict library 38
The internals of the interval framework 38
4.7 Basic design decisions 40
4.7.1 Managing process information 40
4.7.2 Startup 42
4.7.3 Reaction on finished events 43
4.7.4 Identification and assignment of incoming data 46
4.7.5 Treatment of unexpected messages 48
4.8 Data structures 50
4.8.1 RPI specific process data 50

4.8.2 RPI specific request data 54
4.8.3 The envelopes 55
4.8.4 Unexpected messages 57

5 Configuration, compilation and installation of the VIA RPI 59

5.1 Requirements 59
5.2 Installation 59

6 Benchmarks 62

6.1 Comparable software 62
6.2 Testbeds 63
6.3 Results 64
6.3.1 Conformance 64
6.3.2 PingPong 65
6.3.3 Broadcast 69
6.4 Conclusion 71

Bibliography 72

List of Figures

2.1 The LAM-MPI stack 7

3.1 The VI Architecture 11

4.1 Structure of immediate data 22

4.2 Time for mem-to-mem copy vs. registration 26

4.3 Tiny message protocol 26

4.4 Short message protocol 27

4.5 Long message protocol 28

4.6 The use of the RDMA areas 34

4.7 Memory stack with registered memory 36

4.8 Available scenarios on memory registration 37

4.9 lrulist_entry: Specific data about a memory area (ssi_rpi_via_interval.c) . 39

4.10 lam_ssi_rpi_via_proc_infos: Basic process information (rpi_via.h) 43

4.11 lam_ssi_rpi_proc: RPI specific process data (rpi_via_proc.h) 52

4.12 lam_ssi_rpi_req: RPI specific request data (rpi_via_req.h) 54

4.13 lam_ssi_rpi_via_envl: RPI specific envelop data (rpi_via_envl.h) 56

4.14 lam_ssi_rpi_cbuf_msg: Unexpected message data (rpisys.h) 57

4.15 lam_ssi_rpi_cbuf: RPI specific unexpected message data (rpi_via_cbuf.h) 58

6.1 Pingpong on cluster 1 66

6.2 Pingpong latency details 66

6.3 Comparison of the VIA RPI and the native M-VIA 67
6.4 Pingpong bandwidth details 67
6.5 Load of the frameworks 68
6.6 Pingpong on cluster 2 69
6.7 Broadcast with 4 nodes on cluster 1 70
6.8 Broadcast details on cluster 1 70
6.9 Broadcast with 4 nodes on cluster 2 71

1 Introduction

1.1 Current state of related works

Nowadays there exist several MPI-libraries for executing parallel programs. MPICH and LAM-MPI have turned out to be the most often used MPI environments. They run on "of the shelf" PCs and they use various network solutions such as Fast Ethernet. Distributed applications require a rapid and reliable exchange of information within the network. Unfortunately the latency and bandwidth of Fast Ethernet is not entirely used. The reason is the use of TCP/IP as the protocol for interprocess communication. TCP/IP does not provide these features due to its layered structure where data copies are performed through kernel space to user space through several layers.
In January 1997 a consortium consisting of Microsoft, Compaq and Intel defined a new standard solving this problem : The Virtual Interface Architecture. This standard defines the architecture of an interface between high performance networks and the application layer in an efficient manner. Many basic approaches for carrying out this standard have derived. One of them, the M-VIA library from scientists of Berkeley University, won out over the others and is specialized for Ethernet Networks. It is an implementation for Linux that emulates VIA for several Ethernet cards and provides programmers with a VIA-API.
Currently two MPI libraries are using M-VIA: MVICH, an MPICH variant created by the developers of M-VIA, and a VIA device for LAM-MPI from the italian University of Parma. By testing both solutions the following results occurred : The LAM-VIA device does not run at all due to mistakes in the official source code and it would only work with

a rather old version of LAM-MPI. For that reason we could not perform any tests. The MPICH variant did work. But it is not a device for usage with other MPICH versions. It is a special implementation. Therefore this solution is also obsolete and does not follow the new MPI standards and functions. As a result for our tests the need for a new VIA device which is documented in this diploma thesis, arose.
As a first step we had to make a decision whether to develop this device for LAM-MPI or for MPICH. We chose LAM-MPI because of it's modular internal structure explained in chapter 2.3 and its excellent documentation for developers. Furthermore LAM-MPI includes useful features such as checkpoint/restart and a transparent use of different networks on runtime. Grid capabilities are going to be released soon. Because of it's modular structure, communication modules are suitable with subsequent versions of LAM. With these attributes LAM-MPI is a secure solution for the future and worth developing the VIA device on its bases.

1.2 Goals of this work

- Developing a VIA device for the new LAM-MPI using the software emulated VIA capabilities of M-VIA.
- Full conformance with the MPI standard realized in LAM-MPI 7.1.
- Obtaining comparable or better benchmark results then MVICH, and the TCP/IP versions of LAM resp. MPICH.
- Creating the device as a pluggable module for an easy installation.

2 LAM-MPI - a high quality implementation of the MPI Standard

LAM/MPI is an open source MPI implementation of high quality that works on most flavors of Unix (POSIX). LAM/MPI provides high performance message passing on small off-the-shelf clusters as well as large SMP machines with high speed networks and even in heterogenous environments. LAM-MPI is full MPI-1.2 conformant and holds a great part of the MPI-2 functionality, such as:

- Mpiexec
- Thread support (MPI_THREAD_SINGLE - MPI_THREAD_SERIALIZED)
- User functions at termination
- Language interoperability
- Dynamic processes (MPI_Comm_spawn)

Other useful features are official C++ bindings, parallel I/O (provided by ROMIO) and one-sided functionalities, guaranteed resource cleanup (Ctrl C means Ctrl C), fast mpirun startup time and a strong debugging support all together with high performance. These features result in a very cluster friendly solution. Points that should be considered are a strong user base and active mailing lists.

2.1 Features of LAM-MPI

The official feature list of LAM-MPI is listed in the followed paragraphs[1] .

- Checkpoint/Restart
  MPI applications running under LAM/MPI can be checkpointed to disk and restarted at a later time. LAM requires a 3rd party single-process checkpoint/restart toolkit for actually checkpointing and restarting a single MPI process - LAM takes care of the parallel coordination. Currently, the Berkeley Labs Checkpoint/Restart package (Linux only) is supported. The infrastructure allows easy addition of new checkpoint/restart packages.

- Fast Job Startup
  LAM/MPI utilizes a small, user-level daemon for process control, output forwarding, and out-of-band communication. The user-level daemon is started at the beginning of a session using lamboot, which can use rsh/ssh, TM (OpenPBS / PBS Pro), or BProc to remotely start the daemons. Although the time for lamboot to execute can be long on large platforms using rsh/ssh, the start-up time is amortized as applications are executed - mpirun does not use rsh/ssh, instead using the LAM daemons. Even for very large number of nodes, MPI application startup is on the order of a couple of seconds.

- High Performance Communication
  LAM/MPI provides a number of options for MPI communication with very little overhead. The TCP communication system provides near-TCP stack bandwidth and latency, even at Gigabit Ethernet speeds. Two shared memory communication channels are available, each using TCP for remote-node communication. LAM/MPI 7.0

[1]This list is exacted from www.lam-mpi.org on 2003-10-11

and later support Myrinet networks using the GM interface. Using Myrinet provides significantly higher bandwidth and lower latency than TCP.

- SMP-Aware Collectives

  The use of clusters of SMP machines is a growing trend in the clustering world. With LAM 7.0, many common collective operations are optimized to take advantage of the higher communication speed between processes on the same machine. When using the SMP-aware collectives, performance increases can be seen with little or no changes in user applications.

- Integration with PBS

  PBS (either OpenPBS or PBS Pro) provides scheduling services for many of the high performance clusters in service today. By using the PBS-specific boot mechanisms, LAM is able to provide process accounting and job cleanup to MPI applications. As an added bonus to MPI users, lamboot execution time is drastically reduced when compared to rsh/ssh.

- Integration with BProc

  The BProc distributed process space provides a single process space for an entire cluster. It also provides a number of mechanisms for starting applications not available on the compute nodes of a cluster. LAM's BProc support supports booting under the BProc environment, even when LAM is not installed on the compute nodes – LAM will automatically migrate the required support out to the compute nodes. MPI applications still must be available on all compute nodes.

- Globus Enabled

  LAM 7.0 includes beta support for execution in the Globus Grid environment.

- Extensible Component Architecture

  LAM 7.0 is the first LAM release that includes the System Services Interface (SSI), providing an extensible component architecture for LAM/MPI. Currently, "drop-in"

modules are supported for booting the LAM run-time environment, MPI collectives, Checkpoint/Restart, and MPI transport (RPI). Selection of a component is a run-time decision, allowing for user selection of the modules that provide the best performance for a specific application.

- Easy Application Debugging
  Debugging with LAM/MPI is easy. Support for parallel debuggers such as the Distributed Data Debugging Tool and the Etnus TotalView parallel debugger allows users straight-forward debugging of even the most complicated MPI applications. LAM is also capable of starting standard single-process debuggers for quick debugging of a subset of processes. A number of tools, including XMPI are available for communication debugging and analysis.

- Interoperable MPI
  LAM implements much of the Interoperable MPI (IMPI) standard, intended to allow an MPI application to execute over multiple MPI implementations. Use of IMPI allows users to obtain the best performance possible, even in a heterogenous environment.

2.2 The Architecture of LAM-MPI

LAM-MPI is separated into two general layers : the LAM runtime environment (RTE) and the MPI communication layer. The MPI layer depends on the LAM layer functions but both layers interact with the operating system because in some cases it is time-consuming to deal with OS functions over the LAM layer if direct access to them is also possible and sufficient. The stack of LAM-MPI is shown in figure 2.1 These two layers are located in different libraries. The LAM layer consists of the RTE and a library providing C functions to use the RTE. The RTE consists of daemons the so called lamds and provides several services such as message passing and process control I/O for-

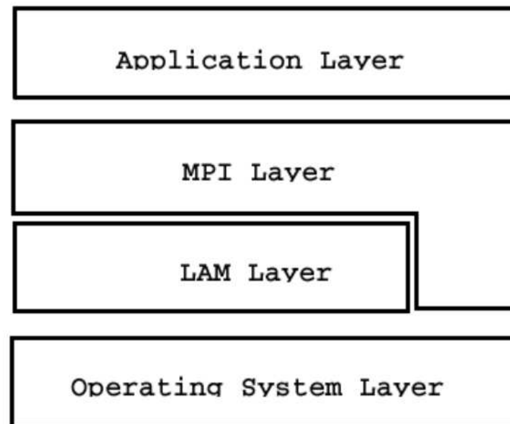

Figure 2.1: The LAM-MPI stack

warding. On each involved node a daemon is running to launch MPI or ordinary programs.

The MPI layer includes two sub layers. The upper one includes the API of the MPI functions and the lower layer provides the interface for various components of the SSI layer. The MPI layer always depends on the LAM layer, but the LAM layer can be used independently from the MPI layer.

2.3 SSI - The System Services Interface

Up to now LAM-MPI used to have a monolithic architecture. Because of its complexity and size, new LAM developers found it difficult to work in the code base, despite of the well structured source code. This fact was also a problem for third party developers. This situation led to a new version of the LAM-MPI architecture with a modular component-based design called System Services Interface (SSI). With that framework it is now possible to develop small and independent modules, which are easily pluggable

into LAM-MPI. The developers do not need to know the internals of LAM. With that it is possible to unite modules of the same component type (for example transfer modules for GM and TCP) into one process. The modules can also be selected manually by the user at runtime. Each module is a self contained set of source code with its own directory structure and can configure, build and install itself. A comparison between the modular version LAM-MPI 7 and the monolithic version LAM-MPI 6.5.9 has shown, that the new version is actually a bit faster.[7]

The following modules have been developed so far :

- Boot : Provides the possibility for launching LAM daemons in different execution environments such as rsh/ssh, PBS, bproc, Condor and Globus.
- Coll : The backend for MPI collective algorithms
- Checkpoint / Restart (blcr) : Allows the checkpointing of parallel MPI jobs
- RPI - Request Progression Interface : Provides the backend for MPI point to point communication for shared memory, TCP and GM

The SSI framework in separated into two parts and administers collections of system interfaces. Its goal is to compose the total system of selected interfaces for each kind of these collections. The user can apply command line arguments for them. That way parameters may be passed to choose, for example the network device for transferring (TCP, GM, ...) at run time. Each module is responsible for its creation mechanism, which requires the creation of the directory structure, configure scripts and Makefiles. LAM-MPI uses the Autotools and Libtool to generate these scripts. Hence the scripts must be accommodated in order to be detected and installed by the LAM framework and they must keep given SSI design restrictions. Moreover each module has to provide a priority value for being able to make a decision which module to choose if several modules exist. Therefore one

step to our RPI device was the creation of the autotool scripts and the creation of directory structure, prescribed in [6]. With this infrastructure source code can be written and tested in a working LAM environment.

3 Virtual Interface Architecture

Rapid developments in networking technology and a increase in clustered computing, drove research studies on communication architectures with high performance protocols. The most often used network protocol is TCP/IP which is optimized for wide area networks. It deals with heterogenous networks and reduces appearing errors by an error correction mechanism. But due to its design it can not take full advantage of all features of current network hardware. So TCP/IP is not the appropriate protocol for system area networks and clusters. For solving this problem the VIA standard was established.

3.1 Design

VIA (Virtual Interface Architecture) is a communication protocol designed to improve network performance regarding latency. The improvement is gained by providing a direct user access to the network interface, without causing any intervention of the operating system kernel and intermediate copies of data. It defines a set of functions and their associated semantics used for moving data into and out of process memory. The design of the VI Architecture focuses on low latency and high bandwidth for exchanging data between processes with a minimum of CPU utilization. CPU utilization is minimized by avoiding interrupts and context switches whenever possible. Concurrency of the communication interfaces is managed without intervening the operating system.

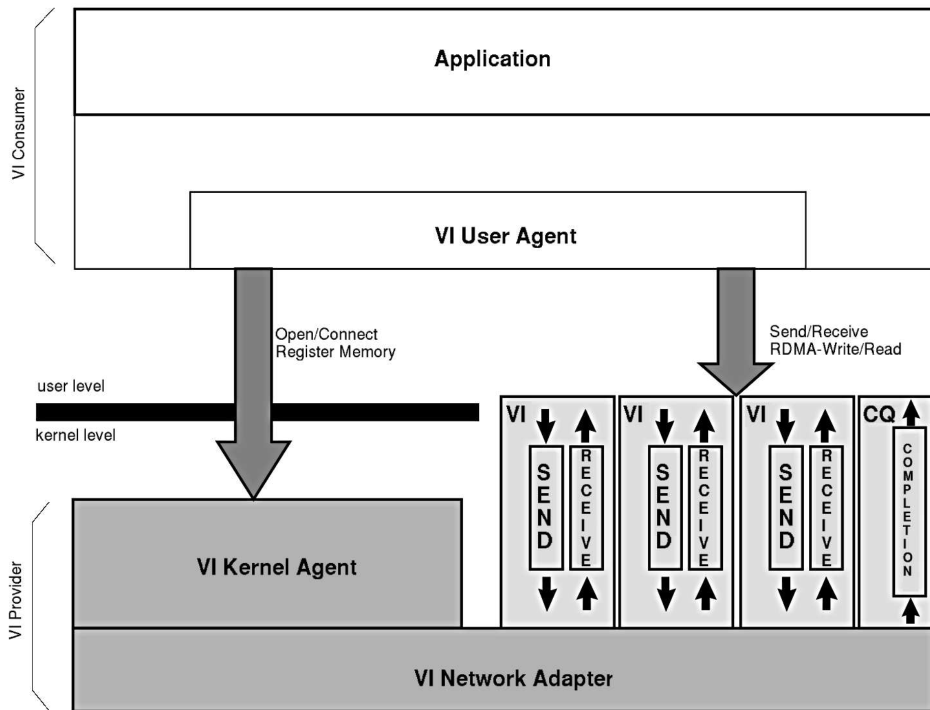

Figure 3.1: The VI Architecture

The VI Architecture comprises four basic components: Virtual Interfaces, VI Providers, completion queues and VI Consumers. These components are shown in figure 3.1.
The interface to access the network hardware, is called the Virtual Interface (VI). It implements the mechanism to form a connection endpoint to another process. VIA is connection oriented, i.e. a connection over a VI has to be established in order to exchange data. A connected pair of VIs can setup communication channels to allow bidirectional point-to-point data transfer. Using the VI channels avoids the protection check of the operating system on the critical path of data transfer.

The VI Provider is a set of hardware and software components to administer instances of Virtual Interfaces. The VI Provider consists of a network interface controller (NIC) and

a Kernel Agent. The VI NIC implements the Virtual Interfaces and Completion Queues and performs data transfer functions directly. The Kernel Agent as a part of the operating system is usually a driver that performs the setup and includes essential functions for resource management. These are vital for maintaining a Virtual Interface between VI Consumers and VI NICs and include the creation/destruction of VIs and the memory management.

The VI Consumer is generally composed of an application program and an communication facility within the operating system. It represents the user of a Virtual Interface. VI Consumers send and receive messages by posting requests to send and receive queues in the form of descriptors. Descriptors are data structures containing all the information about the data transfer. To these belong for example the source and destination address and the length of data to be transfered. The moment the descriptor is posted, the NIC is informed that new data must be transfered. VI Consumers access the Kernel Agent using system calls of a provided VIA API. The requests are asynchronously processed by the network interface controller (VI Provider). When the requests are completed, they are marked with a status value called doorbell. Afterwards VI Consumers can remove these descriptors from the queue and reuse them if necessary. In addition the VI Consumer has the opportunity to combine the descriptor completion events of multiple VIs into one single queue called completion queue. In that way only one queue has to be checked if more then one connection to several processes exists.
There are several possibilities how to get notified if an event on one of these queues has occurred. There exist blocking and non blocking polling mechanisms and the option that a self defined callback function is called if a transfer has finished or when new data has arrived at a VI.
Another key feature of VIA is Memory registration/pinning. VIA determines that only data from registered memory is able to be transfered. Any memory page that is registered with VIA is kept pinned to the same physical memory location until the memory is deregistered

by the VI Consumer. The advantage of transferring data directly between the buffers of a VI Consumer and the network results in saving additional copies of data. Such zero-copy communication protocols help to improve the performance of the communication.

3.2 Basic concepts

Pinned buffers and memory handles Each memory region, that serves as a send or receive buffer must be pinned completely. This is done by the VIA function `VipRegisterMem()`. This function requires the start address and the length of the buffer and returns a handle belonging to this area. This handle is used on every data transfer the memory area is participated. In order to unpin this buffer the function `VipDeregisterMem()` has to be applied.

Descriptors A transfer request is always described by a descriptor. It is a data structure that contains all needed information to process the request. It includes the transfer status, source and destination address, the length of the data, to be sent away and additional values depending on the communication principle. It also contains the memory handle.

Immediate data Each RDMA descriptor has an 32 bit immediate data field and an immediate data flag. If this flag is set, it indicates that there is valid data in the immediate data field. In a send descriptor, it adverts that the data in the immediate data field has to be transferred to the corresponding receive descriptor on the remote end of the connection. This is useful to transfer additional information. With this feature notification on the remote side with RDMA is possible. As required for Send/Receive, it requires pre-posting of the descriptor on receiver side.

3.3 Communication principles

Send/Receive It is the most common transmission model. The sender has to specify where the data to be sent, is located. The receiver specifies the location where received data should be placed. This is done by setting address and size values in descriptors. By posting the descriptor into the send queue on sender side and into the receive queue on receiver side, the data transfer is initiated. It is important, that the receiver pre-posts its descriptor before the sender does it.

RDMA Write RDMA write means, a process can write into the memory of another process, without preceding synchronization with that peer. With VIA, writing is only allowed in registered memory. That is why the transfer buffers have to be registered by each of the peers. The Sender must specify the source and destination buffer and the corresponding memory handles. This implies hat the receiver has informed the sender of the location of the destination buffer and its memory handle. If the data arrives, no descriptor on the receiver side is consumed. The receiver does not get any notification that the transmission has completed.

3.4 Notification

Polling In order to get informed about arrived data, VIA defines blocking and none blocking functions (VipSendDone,VipSendWait,...) to check send/receive or completion queues for a completion entry. If an entry was found, it returns the address of the corresponding descriptor.

Notify functions Another option is applying notify functions (e.g. VipSendNotify) to a send, receive and completion queue. After a descriptor has marked as done this function

requests that a designated handler is called. This handler will be invoked with the address of the completed descriptor as a parameter.

3.5 M-VIA -A high performance modular VIA for Linux

VIA can easily be integrated into hardware and software. The first VIA network cards were released by hardware producers. Without special VIA hardware it is also possible to emulate it as a software library. An example for that is M-VIA - an implementation of the Virtual Interface Architecture (VIA) for Linux. M-VIA was developed as part of the NERSC PC Cluster Project. M-VIA coexists with traditional network protocols such as TCP/IP. It has the distinction of low latency and high bandwidth because of the more lightweighted VIA communication protocol, and it provides many common Ethernet cards. The modular design of M-VIA offers a simpler method to port it to new hardware. The use of techniques such as fast traps results in a higher performance. M-VIA fulfills every Intel conformance test for VIA.
M-VIA abstracts the Kernel Agent into the M-VIA Kernel Agent and one or more M-VIA device drivers. A VI Kernel Agent combined of the M-VIA Kernel Module and the M-VIA device driver for the hardware. The M-VIA Kernel Agent performs privileged operations on behalf of the VI Provider Library and assists the drivers with operations that requires operating system support.It is divided into the following functional components that are device independent:

- Connection Manager for establishing logical point-to-point connections between VIs.
- Protection Tag Manager, that allocates, deallocates, and validates memory protection tags (Ptags).

- Registered Memory Manager that supervices the registration of user communication buffers and descriptor buffers.
- The Error Queue Manager provides a mechanism for posting asynchronous errors by VIA devices

An device driver is an abstraction of a VI NIC, and registers itself with the M-VIA Kernel Agent. The driver informs the Kernel Agent of its capabilities such as direct VIA support in hardware, its native MTU size, the maximum number of descriptors that can be queued for transmission, and so on. It also registers device specific functions that are used by the modular managers of the Kernel Agent layer. The developer of the M-VIA device driver has the option of overwriting all of the default functionalities provided by the M-VIA Kernel Agent layer. For example if a device which provides native VIA hardware support, uses its own mechanism for registering memory, it may completely replace the Registered Memory Manager with an implementation of its own. The ability to override all of the default functionalities of the M-VIA Kernel Agent layer should allow any natively supported or software emulated VIA device to be ported to M-VIA.

M-VIA contains a single VI Provider Library. This library is interoperable with native hardware and software VIA devices that were developed within the Modular VIA framework. M-VIA Kernel Agent drivers specify whether ioctl system calls or fast traps are used by the VI Provider Library to call time-sensitive VI Kernel Agent services. M-VIA device drivers also specify whether the VIA Doorbell mechanism is supported directly in hardware as a true memory mapped doorbell or whether it should be emulated with a fast trap.

The M-VIA user library offers nearly all given functions defined by the VIA standard. It implements all transfer principles except RDMA Read.

4 LAM-RPI for M-VIA

4.1 Overview

RPI is an abbreviation for Request Progression Interface, and Request is the appellation for a basic description of a point to point data transmission. All MPI communication primitives, for example `MPI_Allgather`, are disassembled into such point to point transmission requests by the MPI layer of LAM. The RPI has to care about the handling of this requests.

In this chapter, we describe the main work we have concentrated on - the design and implementation of an RPI that uses the advantages of M-VIA.

The chapter is organized as follows : First, it is necessary to give insight into the interface between LAM and the RPI to understand the demands to an RPI. Section 4.3 explains which basic capabilities that are provided by VIA for transmitting data, we used, and why. Section 4.4 is about the differences we made for messages, depending on their size. It contains a brief description of the protocol that is used for each kind of messages. Next, we give a brief overview over to the main request progression engine of LAM and the details that are specific to the VIA RPI. Another very important issue, if using VIA for transmitting data, is the management of pinned memory. Everything about this can be found in section 4.6. What follows in 4.7 are some more details regarding to specific problems that had to be solved, and design decisions related to this. At the end, there is a brief listing of the main data structures this RPI implementation uses.

4.2 The interface to the LAM framework

An SSI module is represented by a public data structure named `lam_ssi_rpi_<module_version>_t`. It mainly contains the function pointers to the open, close, query and init functions of the module. Every SSI module is addressed through these functions. Furthermore, the return value of the init function is another data structure (`lam_ssi_rpi_actions_<rpi_module_version>_t`) that represents the special kind of SSI. For example, for an RPI SSI a structure is returned that again contains pointers to functions. These functions that are described in short by the following lines give LAM the access point to the basic functionality of the RPI. After this, the two most important data structures are introduced because they are necessary to understand further explanations.

4.2.1 RPI API functions

- `lsra_query`: If the RPI is selected, this function is invoked and returns a data structure that contains the pointers to all the necessary functions for using this RPI. It also returns the maximum and minimum thread level that is supported by this RPI and an absolute priority value for it.
- `lsra_init`: This function gets an array of structures filled with information about the attended processes. With this information it initializes the RPI sub-layer and the corresponding processes by establishing these connection between the processes. This is achieved by calling `lsra_addprocs`.
- `lsra_addprocs`: Each RPI has its own list of processes. When one ore more processes are added by MPI functions such as `MPI_Comm_spawn`, this function is invoked with a process list of the new ones in order to introduce them to the RPI.

After that the RPI updates its own internal list and establishes the connection to the already existing processes.

- `lsra_finalize`: This function is called by `MPI_Finalize()` to clean up data structures of the RPI and to disconnect each connection to the remote nodes. It is always called at last, after all communication has finished.

- `lsra_build`: This function is the first part of initializing a request to optimize persistent operation, It builds a RPI portion of a request that is preserved over multiple invocations and is called only once.

- `lsra_start`: After adding a request to the progression list, this function is started. It initializes RPI dependent aspects of a request in order to prepare this request. Having executed, the request is in the active state and ready to be advanced.

- `lsra_advance`: It gets a list of all request, which are in the active state. The function tries to advance all requests as far as possible and as allowed by the RPI. It takes a lot of time and effort to implement this function.

- `lsra_destroy`: This function removes the request of any pending lists and sets all memory free dedicated to the request. It cleans up the portion of a request.

- `lsra_alloc_mem`: For fast message passing, pinned memory is allocated by this function. It works also as backend for MPI_Alloc_mem(). If no special memory is needed, this function stays unimplemented.

- `lsra_free_mem`: This function deallocates the special registered memory.

- `lsra_iprobe`: This is the public interface for `MPI_Iprobe` which does not return a request to the user.

- `lsra_module_open`: This function is invoked to open the RPI. If necessary it serves as a part of RPI initialization apart from `lsra_init()`.

- `lsra_module_close`: This function is called when the module is closed. It is always the last function that is called in this module for the duration of the process.

4.2.2 Basic data structures

The two main data structures, an RPI has to juggle with, are `struct _proc` and `struct _req`. The first one encapsulates all information that are associated with an MPI process, such as its state or its identification (`struct gps` [8]). The second type contains everything that is necessary to specify and work up a request. This includes, for example, the values that where specified in the parameter list of an MPI_Send or MPI_Recv, the state of a request and a pointer to the instance of type _proc that represents the peer process. A detailed listing and description of this, and more data types that are relevant to an RPI, can be found in [8]. An overview over the supplements to this data types that are specific to the VIA RPI can be found at the end of this chapter, in section 4.8.

To keep all explanations a bit more simple, there has to be a basic definition. Whenever is spoken about ”a _proc” or ”a _req”, it has to be understood as an instance of one of those types that represents a definit process or request.

4.3 Transmission principle

4.3.1 Send/Receive vs. RDMA-Write/Read

As mentioned yet (3.5), there are two different principles for transmitting data from one process to another. The VIA specification calls them RDMA-write/RDMA-read and send/receive. This appellation may be a bit confusing because the transport of the data is the same in both cases. Between nodes, the standard protocols that are supported by the

network hardware are used, and inside of a node, direct memory access is used. So neither the one nor the other principle bothers the CPU; and in terms of transmission velocity neither the one nor the other principle is better. The difference is more or less only the user interface. Whereas send/receive involves both sides of the data transfer, RDMA-write and RDMA-read are what can be understood as the onesided operations put and get.

Among other things, the RPI has to deal with unexpected messages. Data has to be received without having a matching receive request. That means without knowing the location and size of the final target buffer for the incoming data. So it is not possible to post a matching receive descriptor because this had to to specify this values. In case of long messages the send send/receive principle could be uded because the sender waits until the receiver informed him about the target address. For other kinds message, what means envelopes as well as tiny and short messages, sends and receives could be used if there would be a set of temporary buffers with a fixed size, at least large enough to hold the biggest possible message. For every of this buffers, a receive descriptor could be prepared properly because of the defined and and never changing start addresses and sizes of these buffers. But since messages may often be much smaller than the size of these buffers and pinned memory is the probably most precious manor within a VIA program, it is an adverse way to use such buffers.

So let's have a look on the read/write principle. With this, there is the problem that the receiver has to find out whether, when, and where data has arrived. The advantage that only one side is involved in a transfer, would solve the above described problem for all kinds of protocols (Details in section about memory management (4.6.1)). But this seems to be nonrepresentational, due to this notification problem. In case of sends and receives, notification for finished data transfers is clearly regulated by a flag inside the descriptor. This ”done” flag is triggered to 1 by the VIA provider if a receive has been completed. All techniques of the VIA user interface that provide notification of incoming messages

(Wait/Done functions or notify functions) are, at the end, based on checking this flag. If using an RDMA-write, there is no receive descriptor and as a result no ”done” bit.

But there is a way to solve this problem and make use of the advantages of one-sided RDMA writes because of immediate data.

4.3.2 Immediate data

Within a descriptor, there is a field of 32 bits, called Immediate data. Whenever a send or a write descriptor is processed with immediate data feature enabled, these 32 bits are sent together with the actual data. On the peer, there has to be a receive descriptor posted that does not has to specify a receive address and length but with the immediate data feature enabled too. If immediate data, together with some message data, arrive, the immediate data are copied into the equivalent 32 bit field of the receive descriptor, and after all data are in place, the descriptor will be marked as done.

By using immediate data, we kill two birds with one stone. First, we are able to get notified of incoming messages, even if using RDMA writes. Second, we can use these 32 bits for transmitting additional information that is necessary for a working protocol. The following is only a brief overview over the structure of these 32 bits as we used them. More about this, in section 4.7.

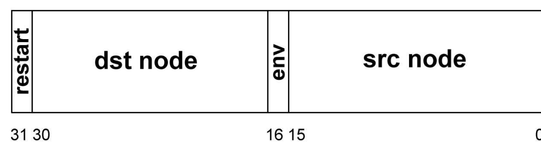

Figure 4.1: Structure of immediate data

- `src_node`: Number of the node that sends the data.

- `dest_node`: Number of the node that shall receive the message.
- `env`: If set to 1, the received message is an envelope or a tiny message.
- `restart`: Indicates that the sender has again begun to write to the top of the RDMA area. See section 4.4 for details.

4.3.3 Empty receive queue problem

An incoming message that carries immediate data needs a descriptor posted to the receive queue (RQ). An asynchronous error is raised if there is none. Usually this ends up in not only losing the message, but also in losing the connection and crashing the program.

So because the RPI must be prepared to receive messages anytime, it has to be ensured that the RQ of every VI is never empty. The most secure way to achieve this is surely some kind of flow control that keeps every process informed about the state of the corresponding RQ of every other process and forbids sending a message out, if there is no receive descriptor available. If RDMA-read would be implemented in M-VIA, this would not be very problematic and should also be not that time consuming. But since M-VIA does only support RDMA-writes, this is a bit more expensive regarding to programming work and run time. At least, based on the following considerations and sanctions, we decided not to implement such a flow control mechanism.

First, VIA does only allow point to point connections between two VIs. That means only one definite process is able to consume receive descriptors of another definite process. Since we recycle and repost every finished receive descriptor directly after handling the corresponding arrived message, the only case in which a RQ could become empty is when one process floods another process with many small messages while this one is occupied with, for example, calculation. Now with the assumption that no meaningful ”real world” MPI program would do such fast and continuous message sending without expecting any

response, it should be save to do without any protection mechanism, above all, to save worthy time.

4.4 Message classification

As described above, VIA offers the possibility of reading and writing directly from and to user space memory, even of remote processes. This opens a true chance to speed up internode communication. Since there is no use of making extra copies from user buffers to system buffers, the time for doing this can be saved. One of the rubs in this principle is that the sender has to know a memory address of the remote node where he is allowed to write to, or better where the remote process does expect the data.

So there are two basic principles to deal with that problem. The first is to define one or more fixed areas inside the memory where another process is always allowed to write to. The address of this or these is needed only once to be transferred to the other process. After this, the remote process may manage the space itself, heeding to a defined protocol. The second is to send the address of the target buffer to the sender just before each single exchange of a message.

The first principle necessitates only one extra copy of the data on the receivers side - out of this area and into the target buffer specified within a request. Furthermore, this requires a set of rules and some kind of handshake protocol to ensure integrity of the data within this area. The sender has to know whether or when and where within this area he may write to without overwriting earlier messages. In the issue, the receiver has to know where and when he may read the next message from. Also things like free space management and availability of other resources such as descriptors has to be minded.

In the second case, additional time has to be accepted because of the extra message containing the target address, and not at last because of the synchronization of the two com-

munication peers that is caused by this extra message. Of course, waiting for another process could be a not little waste of time. Into the bargain, we have to add the expenses of time, caused by the registration of source and destination memory. But for all that, this is a suitable principle for exchanging long messages. First we do not know the size of the largest message of a program. So we can not use a pre-registered area with a fixed size, and dealing with packets of fixed size would also be limited by the number of the buffers, or would, after all, require synchronization too. A protocol would be required that cares about stopping and continuing communication depending on the availability of free buffers. Second, and much worse, the time for the extra copy of a large message is huge but the time for registration could be bated by using ”lazy deregistration” as described in section 4.6.3. To get an impression of the differences, regarding to velocity, of registration and mem-to-mem copying, see figure 4.2.

However, the fist principle is our choice for smaller messages. The disadvantages of an extra message and a process synchronization carry no weight the larger a message is, but for small messages it is fatal. Furthermore, we decided to divide ”not long” messages again into small and tiny messages. For small messages we decided to use the lazy deregistration engine on the sender side because, depending on how often a special MPI program reuses the same memory for communication, it might be faster because it works with only one extra copy and, with high probability, without any registration. The drawback of sending out of user memory is the need of a two-message protocol The values for the identification of a message (envelope) has to be sent apart. This is explained in detail in section 4.4.2. Therefore, we decided to add a tiny message protocol that needs only one message because of copying both data and envelope into one registered send buffer. As said, making extra copies is much slower than registration for large data, but it is much faster for tiny messages. This makes again the separation of tiny and short messages suggestive. The concrete sizes that divide messages into these three classes are not static but may be changed at runtime using the default way of passing arguments to an SSI module within LAM. (see section 5)

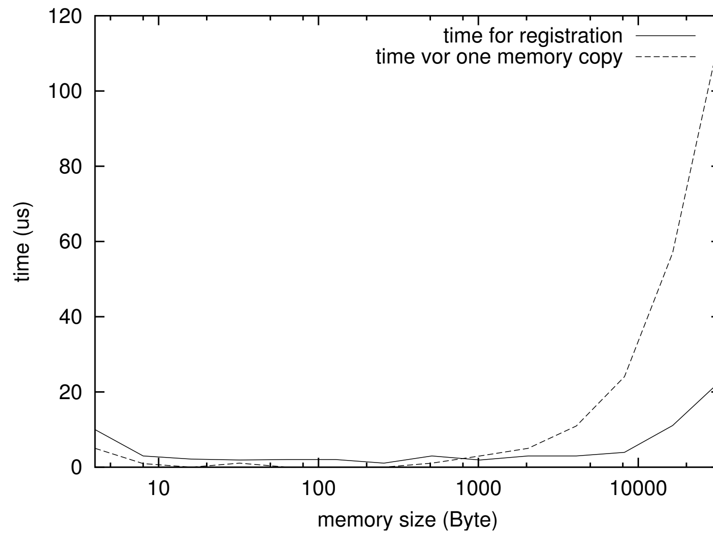

Figure 4.2: Time for mem-to-mem copy vs. registration

4.4.1 Tiny protocol

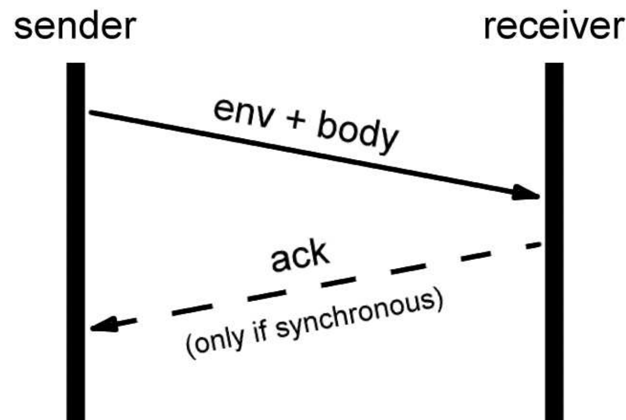

Figure 4.3: Tiny message protocol

As obvious in figure 4.3, the tiny message protocol is quite simple. Envelope and data are both copied into one pinned buffer and transmitted to the receiver in one packet. In case of a synchronous message (MPI_Ssend), an acknowledgment has to be sent back to inform

the sender that all data has arrived. An acknowledgment is eventually an envelope too, except a special flag that is triggered to one.

4.4.2 Short protocol

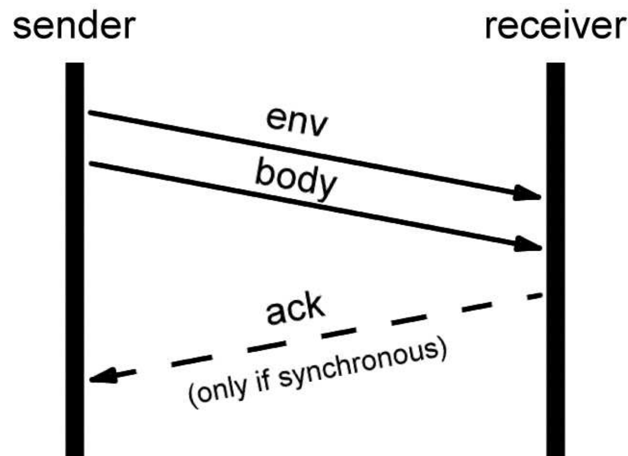

Figure 4.4: Short message protocol

In case of short sends the message buffer from the _req is directly pinned and used for the data transfer. Therefore, the envelope and the message body has to be sent separately. Now when the body arrives, the receiver has the problem that he does not know which envelope and with it, which request it belongs to. As each process has its own RDMA area within the memory of all other processes for writing to, we are able to make the assurance that messages arrive in the same order as the sending process has sent it. So if we always send the body directly after the envelope, the receiver can be sure that the body belongs to the last received envelope.

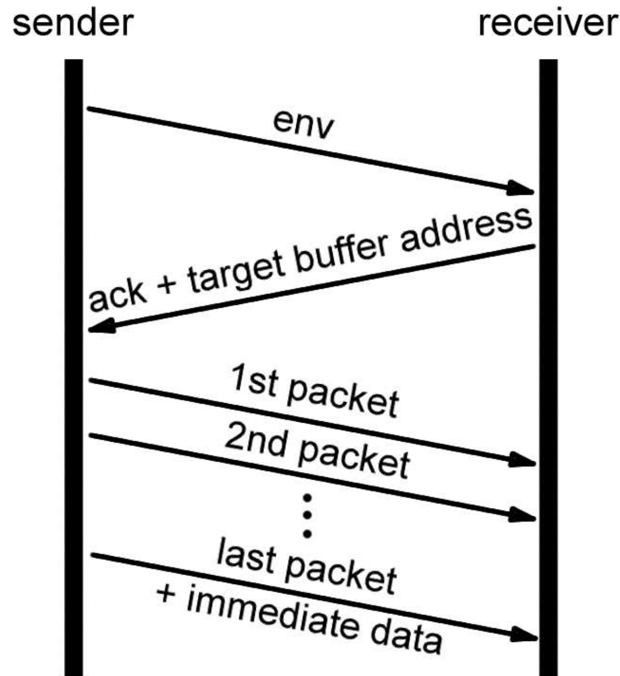

Figure 4.5: Long message protocol

4.4.3 Long protocol

In case of long messages, time consuming copies into extra transfer buffers should be avoided. This is reached by pinning the user buffers on both sides. The long message protocol is subdivided into an exchange of information and the actual data transfer. At the first part the sender sends an initial envelope (SYN) in order to announce that a long message will be sent. The receiver answers with an envelope (ACK) including buffer address and buffer handle of the receivers user buffer. With this information the sender is able to push the message body over. The transfer size for every descriptor is limited by M-VIA. If the limit exceeds, the message is transferred in single pieces. After finishing the RDMA write of the body, sender and receiver need to get notified about the end of this process in order to mark the request as finished. This could also be accomplished by

sending an extra envelope after having sent the last package. But by using immediate data within the last send descriptor this time can be saved.

4.5 A requests life

This is just an overview over the main request progression engine - the steps that every request has to take from its creation to its completion. Since all global function names of the VIA RPI start with ”`lam_ssi_rpi_via_`”, we will hence replace this prefix with $\sim$ whenever we refer to a special function for reasons of simplification.

4.5.1 Building

If a new request is build, the LAM MPI layer calls lsra_build to give the RPI the opportunity to do or change something before the request will be started. Usually, and as the VIA RPI does it too, this is used for the allocation of the memory for the RPI specific request data `rq_rpi` and doing some basic initializations. Since allocating memory needs system calls, it may be faster to avoid calling `malloc` as often as possible. To implement this, we used the memory pool functions that are sopported by LAM. With these, the currently allocated memory for `rq_rpi` will not be given back to the system but stored in a list. Now if lsra_build is called again for a new request, we may just take a pointer out of this list, instead of troubling the system.

4.5.2 Starting

The function that is called directly after this is lsra_start. This is, where the memory for the envelope is allocated as described in section 4.6.2. Furthermore, this is the point where the ways of all request are separated from each other depending on its message size.

This happens by assigning different function pointers to `cq_send_advance_fn`. Also depending on the message size and with it, the protocol that has to be used, the message buffer is pinned using the dynamic memory management (section 4.6.3). If a request is a receive and a matching envelope is found in the unexpected received messages, it will be advanced and maybe finished already here.

4.5.3 Processing

lsra_advance is called repeatedly. All requests that are still in progress are tried to be finished or at least continued here. To avoid many tests for destining the state of a request at this time critical point, each request has two function pointers `cq_recv_advance_fn` and `cq_send_advance_fn`. Whenever something is done with a request, this pointers are updated to point to a function that encapsulates the steps that has to be done whenever the progression of the request can be continued. For example, if the initial envelope of a long message is sent, nothing more can be done until the receiver has not sent the acknowledge back that contains the address of the target buffer. So the progression of the request is stopped for the moment but the receive advance function pointer is set up to refer to a function that cares about the initialization of the send of the body as soon as the ACK has arrived.

When lsra_advance is called, it steps through all requests and executes its send advance function, in case of a send, and queues the request up to the pending receive queue of the source _proc, in case of a receive. Further, `~via_advance` is called to check for any completed VIA events using `VipCQWait` if we are allowed to block, or `VipCQDone` otherwise. Every finished receive event ends up in `~recv_event`, which cares about storing the message into unexpected message buffers or into the one that is specified by a matching known request. The receive advance function will be called, the read pointer of

the RDMA area will be updated and the completed receive descriptor will be reposted to the receive queue of the VI.

For every completed send event it is a must to call the callback that has been registered for it in order to do some clean up. Now after getting a send descriptor back, a new send operation will be started if there is waiting any in `~send_queue`. This queue stores requests with a defined send advance function to perform a send operation. It is filled whenever a try of sending data failed due to a lack of send descriptors.

Just a word about leaving lsra_advance. If blocking is allowed, as much events as has been occurred are handled but it is waited for at least one request that has been marked as done. A finished request is recognized by checking whether `lam_ssi_rpi_via_done` is greater than zero, because this value is increased whenever the progression of a request has finished within either a send callback or one of the special advance functions. If lsra_advance is called because of a blocking request a set `lam_ssi_rpi_via_done` is the permission to leave the function but as long as there are fineshed events, these will be handled. If blocking is not allowed (MPI_Isend, MPI_Irecv), lsra_advance will return even if nothing happened on the NIC. On the other hand, if messages has arrived or sent, all of such events will be handled before lsra_advance returns, even in nonblocking mode. This implicates that sometimes a call to MPI_Isend or MPI_Irecv needs a bit more time than he had to, but the send itself is not slower, and other requests can be closed, and its resources freed, earlier.

4.5.4 Destroying

After reaching the done state, each request will be destroyed. The part of this work that the RPI has to do is defined in lsra_destroy and is nothing more than giving back the envelope buffer, removing the _req from any pending receive queue, and in the end giving back the RPIs portion of the request (`rq_rpi`), again by using the LAM memory pool functions.

4.6 Memory management

In order to exchange data with M-VIA, it is required that the source and destination buffers are registered. Because of the time consume of registering and the limit of possible pinnable memory, an extended memory management that meets these restrictions is needed. The VIA RPI distinguishes three parts of memory management: First is the supply of a memory area for incoming envelopes as well as tiny and short messages. Second is the availability of buffers for outgoing envelopes and tiny messages and the last requirement is the management of long and outgoing short message buffers. The goal is to save the time for registering as often as possible. This is realized by reusing memory that is already registered.

4.6.1 Incoming envelopes plus tiny and short messages

As accounted in section 4.4, envelopes and tiny messages as well as short messages are received into extra pinned memory and copied to its real destination after this.

For the matter of the amount of this pinned memory it is not useful to reserve more bytes than the number of receive descriptors multiplied by the largest possible size of short message. This is enough to ensure that the VIA RPI can not run out of pinned bufferspace as long as it does not run out of receive descriptors. In case of using many buffers of fixed size, it is also not suggestive to have less memory than descriptors because if the one value is tuned well, than the other is it automatically too. But the use of such buffers is not very economic. Every message with only a few bytes would occupy a full buffer. Much memory would be unused but could also not be used for buffering other incoming data. So we decided not to use several small buffers but one bigger area of RDMA-able (pinned) memory.

In order to coordinate the access to this RDMA area, a read pointer is used to indicate where the next received message can be read from, and a write pointer defines where the free space within the RDMA area of another process starts. It is easy to understand that a process has to have separate write pointers, one for every other process, but it is also necessary to manage separate read pointers and even separate RDMA areas. If a process had only one RDMA area and all others were allowed to write to this, its write pointers could not be kept consistent, except using a lot of management communication that would transform a real network to something that rather seems to be like a bus. Of course, having multiple RDMA areas implicates the need of having multiple read pointers - one for each.

Additionally it is possible to tune the size of this area by passing command line arguments to mpirun. This is useful for programs that primarily use tiny or long messages. This saves pinned memory and increases the number of processes that can be used without running into trouble because of memory deficiency. Also the performance of sending and receiving long messages could increase since the probability of having to unregister old and register new user buffers is less.

4.6.2 Outgoing envelopes and tiny messages

Envelopes and tiny messages are also placed into separate memory buffers to send them. In order to save pinned memory, the use of a larger pinned area instead of many separate buffers seems to be a suggestive solution for envelope buffers too. But there is a decisive difference to the situation described in the previous section. Outgoing envelopes will not return in the same order as they had been issued. So managing free space within such an area would be much more difficult. Either we risk to mark memory as usable that is still in use of any request or we introduce an additional security mechanism to avoid this. The first may be work for the most programs but the more processes and the more simultaneous messages there are, the more insecure it becomes. We decided to go a secure way. The

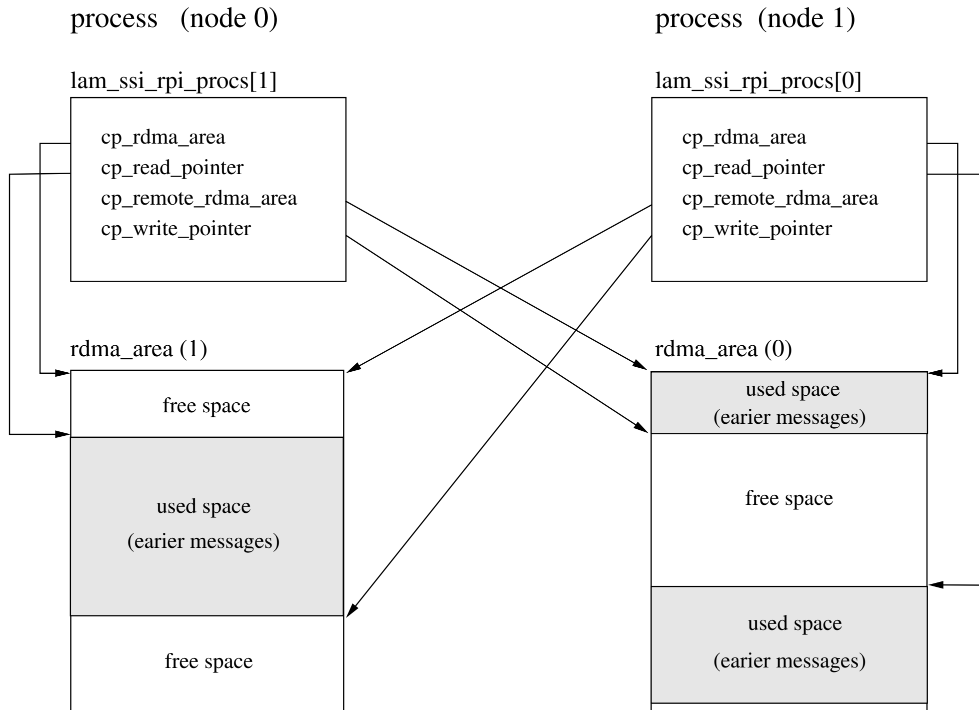

Figure 4.6: The use of the RDMA areas

problem with the mentioned protection mechanism is that it costs time. A sorted list or something similar would be required. Since managing of such a list does not need only little time and querying of an envelope buffer is part of the time critical path, this can not be afforded. On balance, to use several separate buffers seems to be the best solution because every buffer only has to be as large as a tiny message plus some bytes for the envelope.

For that, it is only necessary to manage a list that holds pinned envelope buffers and its memory handles. With any query for a new envelope, the first element of this list is removed and given out. A returning buffer is just attached to the list again. In case the list is empty, a new request for an envelope buffer is handled by allocating and pinning additional buffers, just enough to exploit a whole memory page. This mechanism implies that we will not have less, and also not much more, buffers pinned than we really need.

4.6.3 Long and outgoing short messages

The long message protocol has the property of transferring directly between the user buffers of the participated processes. The registration of these buffers on both sides occurs currently on both sides. This means the total overhead is the time for the registration of these buffers. But registering before the transmissions of data, is still very time intensive.

In most cases an MPI program reuses the same memory location more then once. The solution to save registration time is memorizing areas of the process memory that are already registered. If the need of registration arises, it has to be checked, if the required buffer is already pinned. If it is, any further registration is not necessary.

M-VIA provides special functions to register and deregister memory. The register function VipRegisterMem() obtains the address and the length of the buffer and returns a handle for this buffer, that other VIA function need to use this buffer. Thus for every data transfer a memory handle for the concerning memory area is required. Each handle must be saved with the other information of the area such as start address and length of the buffer, in order to reuse it, if the corresponding memory area is needed again.

A disadvantage of this behavior is the following. If only a part of the required area is pinned, this part is not able to be enlarged. The whole area must pinned again in order to obtain a valid memory handle.

Usually every memory management framework uses a hashtable for saving and finding pinned memory pages and additional parameters. This is only possible, if memory pages of the same length are used. There start address would be the index for the hash value. M-VIA does not use any pages, but operates with different sized areas. Thus a hashtable is not suitable for us, because it only finds areas with the same start address. Figure 4.7 shows an problematic example:

The grey range is the already pinned memory. The black range is the requested memory located in a pinned area. Pinning this area is not necessary. If this grey area has been saved

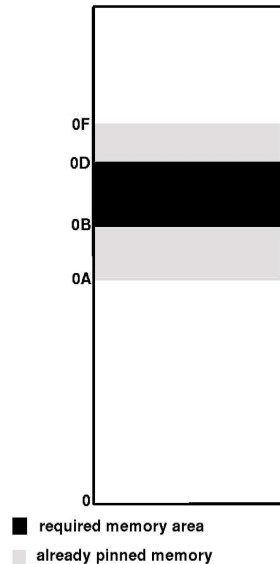

Figure 4.7: Memory stack with registered memory

in a hashtable with the start address as hash value, the black area could not be found due to the different start addresses. Preventing this situation is possible through the use of a data structure for searching sub areas too. Lists and search trees are suitable data structures that could be used.
The decision for a weight balanced tree (AVL tree) was taken because it is almost as fast as a hashtable for less than 3000 entries. It is improbable that more than 3000 long message buffers are registered concurrently, which does not have any intersections. We have also tested a hybrid solution, where areas are saved in a hashtable and in a tree. First the search framework uses the hashtable and if it does not find anything, it uses the AVL tree. Tests with various MPI programs have shown, that this solution has an additional overhead for the hashtable inquiry of 2-4 μs.

If a memory area has to be registered, it is possible that other areas are already pinned and interleaves the new area. There are three main scenarios for this situation shown in

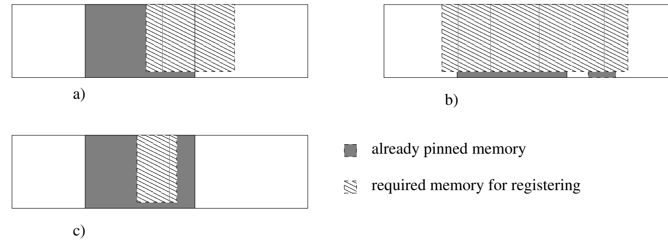

Figure 4.8: Available scenarios on memory registration

figure 4.8. Example *a)* shows partly interleaved areas. In *b)* the area interleaves at least one already registered memory completely. The area in *c)* is a part of an already registered area. In the last case the handle of the greater area can be reused and makes registering unnecessary. In the other cases, reusing the existing handles is impossible, because a handle for the whole area is needed. A combination of several handles from parts of the area is not supported by the VIA standard. The search tree framework must defines a comparison function that explains whether a searched value is lower, equal or greater than the value of the leaf, actually visited by the search tree algorithm. For simplification matters that we are going to take the compared areas are designated as a and b. The start address of an area is designated as sa(area name) and ea(area name) as the end address of an area. In our case the comparison function has to differ between three cases.

- a is less than b if sa(a) < sa(b) and ea(a) <= sa(b)
- a is greater than b if sa(a) >= ea(b) and ea(a) > ea(b)
- in all other cases a and b are equal

The last case means that one of the compared areas includes the other completely or they overlap each other partly. The already registered area can be included by the new area if it is the smaller one. In this situation the older and smaller area has to be removed and unregistered before the new area can be registered.. If we "overpin" another memory area without unregistering it, and at a later time we unpin the older area, M-VIA unregisters the shared parts of the areas too. The newer area can not be used and transfering data out of

this shared part results in failures. But removing of older areas is only possible, if they are not in use anymore because if they are, a process could transfer data from or to that area by using the obsolete memory handle. If this area is unregistered, the handle is not valid anymore. The transfer will fail. In this case the interval framework returns that registering of the new area is not possible at the moment.
All these properties lead to a search tree framework, that supports a data structure with start address and end address of an area as key in the tree. It makes it necessary to have a comparison function that can evaluate these values.

The libdict library

Libdict is a compact, ANSI C library which provides access to a set of generic and flexible dictionary data structures. All algorithms used in libdict have been optimized, and, with one very small exception, are not recursive but iterative. It is released under a BSD style license. Libdict implements for example a height balanced tree algorithm. It provides the possibility to use user defined key structures and a comparison function that determines the relation between two keys. The user can also define functions that are invoked if a key is removed. Because of these possibilities libdict is the suitable search tree framework for us.

The internals of the interval framework

Some MPI functions such as MPI_Alltoall are subdivided into asynchronous operations by the LAM layer. If it is reduced on p2p operations using the long protocol, the RPI always registers the corresponding memory region. In case of several asynchronous operations with overlapping memory areas, additional information about the current status of a memory area must be saved. Status values are saved in a list of structs shown in figure 4.9 and have the following meaning:

```
struct lrulist_entry {
  struct key *key_elem;
  int refcount;
  int lru_ok;
  VIP_MEM_HANDLE memHand;
  struct lrulist_entry *i_prev;
  struct lrulist_entry *i_next;
};

struct key{
  char *addr;
  char *end;
};
```

Figure 4.9: lrulist_entry: Specific data about a memory area (ssi_rpi_via_interval.c)

- `*keyelem`: This is a pointer to the struct `key` containing the start and the end address of the memory area.
- `refcount`: This value counts how many requests refer to this area currently.
- `lru_ok`: It indicates that removing of this area with an LRU strategy is allowed.
- `VIP_MEM_HANDLE`: It saves the belonging memory handle of the area.
- `*i_prev, *i_next`: Pointers to predecessor and successor of this list element.

Each node in the tree contains the `struct key` as the key value and as corresponding data a pointer to one of the elements(`struct lrulist_entry`) of the lru list.
On every operating system the size how much memory can be registered is limited. On that score memory must be released if new one should be registered. The memory framework uses an LRU strategy to solve this problem. New areas are prepended to the head. The

last recently used area is located at the tail of the list. If new pinned memory is required and this area is not already pinned, the framework tries to register it. If it fails, too much memory is registered. In this case the lru list is accessed from tail to the head. If the reference count is zero and lru_ok is true, the element is removed from the list and from the tree. This process is continued until enough memory is freed by retrying pinning after each area removal.

4.7 Basic design decisions

4.7.1 Managing process information

Every process has to maintain information about each other process such as the VIs or the read and write pointers. All this information is encapsulated within a structure named `lam_ssi_rpi_proc`. And this is again part of the LAM structure `_proc`. The full listing and description can be found in section 4.8.1.

Per default, LAM defines a global list of _proc instances (`lam_procs`) but for reasons of efficiency and functionality it is recommended that each RPI has its own local collection of the _procs it is responsible for. For that, the VIA RPI defines `lam_ssi_rpi_via_procs`. A simple array of pointers to the global _procs. This provides access to the information of every process in time of O(1), instead O(n) by using the global process list `lam_procs`. Additionally we avoid inconsistency between global data and the local copies by using pointers. Maybe this has to become real copies when LAM design switches to multi RPI, or there has to be some kind of protection that avoids competitive access to the same _proc by different RPIs. The index for this array is the number of the node. This seems to be the best solution because of the following reasons. There are two situations where it is necessary to deduce from some information that are available to the right _proc. The first situation is trivial. If there is a new send request to

handle, all elements of the identity (gps) of the destination process are known because the request contains a pointer to the _proc of the destination process. So it is easy to find all the information that is needed to perform the send.

The second situation is when data arrive. In this case `VipCQ[Wait|Done]` does only return the vi handle on which the receive occurred. After this, the done receive descriptor has to be dequeued from the receive queue of this vi handle by using `VipRecvDone`. So, additionally to the vi handle, all information within the returned receive descriptor are available. This includes, for example, the length of the received data.

To make further steps within the communication protocol the _proc of the source process has to be found. With the available information it would be possible to manage a hash table that maps from a vi handle to the right _proc but there is a better solution as we think. As described above, we have to enable sending of immediate data. Furthermore we are free to occupy this 32 bits as we want, and having access to the contents of this field is ensured because it are saved in the finished receive descriptor that `VipRecvDone` returned. So it is obvious to save the number of the source node within the immediate data. This is not the only information we transfer within these 32 bits. Figure 4.1 gives an overview.

Let's only make just another annotation regarding to the the node number. At first sight, the rank of a process within MPI_COMM_WORLD (`gps_grank`) would be the better choice. It seems to be unique and would allow communication between processes located on the same node. But there are cases where this `gps_grank` is not unique. In special cases, splitting communicators and creating inter communicators causes the existence of more than one MPI_COMM_WORLD to encapsulate smaller groups of processes. Inter communicators are used to communicate between two of such worlds. With the restriction that the VIA RPI can not care for processes that are located on the same node, the use of the node number as a process identification value is unique. Anyway, the VIA RPI is only sensible and efficient to connect processes that are not communicating with each other through a real VIA enabled network. Furthermore with multi-RPI other RPIs such

as sysv/usysv (shared memory) will care for processes located on the same node in a much more efficient way without the need of adding extra code to the VIA RPI.

4.7.2 Startup

This section gives a brief overview over the initialization and the startup of the VIA RPI.

The first RPI API function that is called is the query function, which checks if the VIA RPI shall run and is at all able to run. Therefore `VipOpenNic` is called already here because this is the easiest way tho check if the M-VIA installation is correct and ready to be used. If not, lsra_query is also the last function that is called. If VIA is all set, an error handling function will be registered to catch asynchronous errors such as broken connections or empty receive queues.

The next is lsra_init. Here, the global memory protection tag and the completion queue are created, and the memory management for long and outgoing short, and for envelopes and incoming short and tiny messages is initialized. Also the queue is initialized that stores send requests that could not be send immediately because of a lack of send descriptors. After this, the processes have to be initialized. This happens by calling lsra_addprocs which is also called if new processes are to be added, and during actions like creating new process groups. So we have to check for that processes that are new and not yet initialized or connected. Since it is only necessary to deal with new processes that have been created on new nodes, the only thing that has to be checked is which process has a node number greater than the actual size of the _procs array. If there are any, the array will be reallocated. After this, every new process, or more precise, all RPI portions of the new _procs that a particular process manages, have to be initialized. For example, the VI has to be created, the RDMA area as well as the send and receive descriptors has to be allocated and pinned, and all static fields of the descriptors have to be initialized. At the end, the

LAM_PRPIINIT flag of the p_mode field of the _proc is set to avoid reinitialization if lsra_addprocs is called again.

The last step within lsra_addprocs is the connection. To connect two VIs using `VipConnectPeerRequest` and `VipConnectPeerWait` the net address of the NIC of the remote node has to be known. Together with the other values within the structure shown in figure 4.10, this is exchanged using the communication functions (nsend/nrecv) of the underlying LAM framework. After this, all information that are necessary to connect to and communicate with another process are available. Again, p_mode is used to indicate that the process is connected to that process that is represented by the actual _proc. The flag LAM_PRPICONNECT is the sentinel for this.

```
typedef struct lam_ssi_rpi_via_proc_infos {

  /* address and handle of rdma area the peer process is allowed write
  char                      *rdmaAreaAddr;
  VIP_MEM_HANDLE             rdmaAreaHand;

  /* my nic address - needed by other processes to connect to me */
  VIP_UINT16             NicAddressLen;
  VIP_UINT8                LocalNicAddress[128];
} lam_ssi_rpi_via_proc_infos_t;
```

Figure 4.10: lam_ssi_rpi_via_proc_infos: Basic process information (rpi_via.h)

4.7.3 Reaction on finished events

To get notified of both send and receive events, the VIA RPI uses a completion queue (CQ) and not the notify functions but `VipCQWait` and `VipCQDone`. This might not

be the most intuitive way but there are several reasons for this decision which shall be explained in this section.

Generally a CQ is a very handy thing if you decided for the polling variant using the ”Wait” and ”Done” functions. If the receive queues (RQ) of all VIs are associated with a CQ, it is not necessary to check every single RQ to find out, if data have arrived from any process. If using notify functions, this advantage is of no object because you don’t have to do any kind of checking. A finished descriptor triggers a defined reaction automatically.

So why not to use notify functions? First let’s have a look onto the case of receives. Notify functions follow an asynchronous principle, what means they are running within a separate thread and may be invoked at every point within the program, even during calculations or other activities outside of the LAM library. This may sound like the easy way to completely efface the empty receive queue problem described in section 4.3.3 because at all times it is possible to react on finished receive events and to repost the descriptor back to the RQ directly after this. But examined in detail, there are more cons than pros. If this immediate reposting shall work, it has to be done inside the notify function. An arrived message requires time, for example, for copying data from the RDMA area to the target or an unexpected message buffer. In case of many messages, the probability that the next notify function will be invoked before the previous one has finished increases, the more and the faster messages arrive. The calling stack could overflow, the danger of lost updates of variables, or similar failures are added and after all, the risk of having an empty RQ resp. a filled RDMA area may be a bit lower but is still there. Additionally handling all receives immediately exalts the probability of having unexpected messages, and so, the probability of having extra actions and additional copies to unexpected message buffers .

Furthermore, some strange errors occurred when we used the notify functions as the following description of the send case illustrates.

There is a major problem that stymies us to reject notify functions for sends too. It occurs when trying to post a send descriptor to the queue and register the notify function for it after

that. Sometimes and not deterministic, M-VIA complains about an empty send queue (SQ) although the previous call to `VipPostSend` returned a success code (`VIP_SUCCESS`).

We actually don't have any idea why this error raises or how to solve or bypass the problem. There are not too much threads and the test programs are extremely spare with using both normal and pinned memory.

Although we favor the use of notify functions at least for sends, this problems forced us to switch to `VipCQWait/VipCQDone`. This is next to no disadvantage regarding to efficiency but a bit more programming expenditure. The notify functions, also called callback functions, allow to associate special actions with a special send descriptor, that means it is possible to define what shall be done after a definite descriptor has finished. This could be, for example, to send the next package of a long message after the send of an earlier package has finished. To make a change back to notify functions as simple as possible, we kept this actions encapsulated within callback functions but we don't use this asynchronous, thread based model for the invocation of these.

We introduced a new data structure `cp_pending_sends` that stores requests that are still in progress and a field `cq_callback_fn` within the _req structure that holds the pointer to the actual callback function. If a process A posts a send descriptor to send something to another process B, the first one stores the send request within the pending send queue of the _proc that belongs to B. Once process A realizes that the mentioned send descriptor is marked as complete, the first _req from the pending send queue of the destination process B is popped to get the request this descriptor belongs to. Than the callback function that is stored in the `cq_callback_fn` list of this requests _req is called. It is always guaranteed that nothing gets in a fuddle because M-VIA always processes descriptors in the same order as they has been posted to the queue.

4.7.4 Identification and assignment of incoming data

Whenever a receive descriptor returns, `recv_event` is called and a decision, what is to be done with the just arrived data, has to be made. A first roughly distinction can be made by inspecting the immediate data field of the descriptor. A set env flag points out that it is an envelope, otherwise it is a body, that is the second part of a message that carries the actual data.

In the first case, some more differentiations has to be made. Because the data are known to be an envelope, they can be accessed as an envelope. For example, the `ce_flags` field of the envelope can be checked. If the C2CACK flag is set, it is the acknowledgment of either a synchronous transmission (MPI_Ssend) or the acknowledgment that carries the target buffer of a long receive request. In both cases, the `cq_recv_advance_fn` pointer of the corresponding request will specify the function that knows what to do next. But how to find this request? Whenever a sender pushes out the initial envelope, he stores the request within the `ve_src_request` field of this envelope and sends it with. The other side just sends this back within the ACK.

If the received data is a regular envelope, there are two possible cases. Either a matching request can be found within the `cp_pending_recvs` list or not. The second case means the receive (e.g. MPI_Receive) that belongs to the send of the other side has not been reached yet within the MPI program. The message is unexpected and `~unexpected_receive_env` will care about it. See section 4.7.5 for details. Expected message is again classified into tiny, short or long messages, or it is just an MPI_Probe. The latter would only cause the update of the requests state. The other cases would end up in calling the corresponding receive function `~(tiny|short|long)_receive_env` that cares for further steps depending on the class of the message.

What is still left over is the handling of message bodies. Long messages are always expected since they use a rendezvous protocol with a hand shake before the body is transferred, i.e. an unexpected bodie is always short. So we may be sure that the content of the data we have to deal with is an unexpected short body if the `cp_bmsg` pointer of the source _proc is not NULL. An earlier receive of the corresponding unexpected envelope has caused the allocation of this buffer. Details can be found in section 4.7.5.

At last let's have a look on expected short or long bodies. Keeping this events apart is a bit more difficult. The message itself carries no information that can be used to find the corresponding request and even the length of the message is fallacious because the last packet of a long message may be shorter than a short one. All we have is the source node that has been extracted from the immediate data, and with it the corresponding _proc. Again a simple list can help (`cp_current_recv_req`). The idea is to store a request in this list whenever an envelope for it has arrived. The first element of this list will be the request the next incoming body belongs to. If this list is kept inside the source _proc the right list can be found if a body arrives (4.7.1)and the only thing that still has to be thought about is the order of the requests in this list and the order in which the bodies may arrive. If only short messages had to be handled, it is easy. Even a list whould be to much. One pointer to the current receive request would be anough because every body arrives directly after the corresponding envelope. Together with long messages, it is a bit more complex. The sender of a long message has to wait for the acknowledgment of the receiver. This fact, together with the possibility of nonblocking sends (MPI_Isend), makes it not possible to ensure that a long body always follows the long envelope (SYN) directly. - Performance would suffer a lot if the time when the sender is waiting for the receivers response would be unused. So during this time other requests may and should be processed. Tiny messages that are handled within this time slot are no problem. The receiver recognizes that it is an envelope and nothing gets confused with the current receive request. In case of multible long messages, only the SYN could be sent for all this messages but after this, it has to be waited for the ACK. All SYNs reach the receiver in the same order as it has been sent,

so all ACKs go back in the same order too, which again ensures that all bodies reach the memory of the receiver in the same order. Therefore, with queuing a new long request to the end of the `cp_current_recv_req` if a SYN arrives, everything is OK. Yet another thing has to be considered if a small request is handled between the send of a long SYN and the receive of the corresponding ACK. Because a short envelope and body are sent one after the other without the possibility of interruption, the small message is both started and finished within the critical period. This is a good condition because the only thing that has to be done is to queue the short request up to the top of the list of current receives, instead of the end.

4.7.5 Treatment of unexpected messages

This section shall give a short summary of the handling of the yet mentioned unexpected messages.

Needlessly to advert that it is possible, within a parallel program, that a send may be executed earlier then the corresponding receive. Because except of the case of long messages, it is not clever to wait until the receiver is ready, but to process the send, not caring about the state of the peer process.

Therefore, the VIA RPI has to be prepared to handle incoming messages, it knows no matching receive request for. This is the unexpected message case. Since the RDMA area has to be vacated as fast as possible, every arrived data has to be copied to other buffers. Without having a request, this can not be the final target buffer but an additional temporary buffer.

There is no restriction made by the LAM framework that prescribes how a specific RPI has to buffer and manage unexpected messages. In spite of this, there is a globally defined structure for storing unexpected message data. It is called `lam_ssi_rpi_cbuf_msg`. Further, there are global functions to deal with instances of this data type. It is possible to

use own data structures and functions for example to perform some RPI specific optimizations but there are two very good reasons for using this global data structure and functions. First, LAM has support for some parallel debuggers such as totalview, which may want to give insight into the saved unexpected messages. If the VIA RPI shall be able to fulfill the requirements for this debugger support, it has to use this `cbuf` structure and functions. The second very important argument of having a common interface for unexpected messages of all RPIs is that LAM/MPI is going to be redesigned to support multible RPIs at the same time. This will open up the support for parallel machines with heterogeneous network connections. In case of MPI_ANY_SOURCE requests within communicators that span multiple RPIs, a common buffering of unexpected messages will certainly be required.

However, there is a possibility to add RPI specific data to `lam_ssi_rpi_cbuf_msg` by defining the type of `lam_ssi_rpi_cbuf`. Having a look to the whole data structure in section 4.8.4 would be helpful to understand the following text.

With unexpected tiny messages, everything is easy. A new instance of `lam_ssi_rpi_cbuf_msg` has to be created, enough memory for the `cm_buf` field to house the body has to be allocated, and finally the body has to be copied over to this buffer. For later identification the envelope is stored too. Before the new cbuf_msg is appended to LAMs internal queue, the `cm_proc` pointer of the cbuf_msg has to be set to NULL. This is a sentinel that the whole message has already been received, and if the right receive request will gets known, it may be completed just during its start phase.

In the case of an unexpected short envelope, only a part of the whole message has arrived. There is no body that could be stored but the receive of the body can be prepared by allocating enough memory for it in `cm_buf`, using the length information from the envelope. After queuing the cbuf_msg up, its address will be stored in the `cp_bmsg` field of the source _proc. This enables discerning the body as a short body when it will arrive, and

the address of an unexpected buffer, we may use to store this body, will immediately be available (4.7.4).

A different case is if the receive request that belongs to this short envelope will be get known (started) between the arrival of the envelope and the arrival of the body. This case is discerned with the help of the sentinel `cm_proc`. It will be unequal to NULL. The receive of the body has to be prepared as this is done for expected short messages. The previously pepared unexpected buffer is not needed any more. It will be freed and `cp_bmsg` will be set to NULL. The receive advance function will be initialized to receive the body, and the request will be prepended to the `cp_current_recv_req` list of the source _proc as described in section 4.7.4

In case a receive request gets known not before both the envelope and the body have been received `cm_proc` will be NULL, `cm_buf` will hold the body, and so all that is needed to mark this request as done is available.

For envelopes (SYNs) of long messages it is again easy because no more data will arrive until the inherent request is not known. As soon as this request is started the target buffer will be pinned and its address will be send to the sender. For now, all that has to be done is just to buffer the envelope and to set both the body buffer `cm_buf` and the `cm_proc` pointer to NULL.

4.8 Data structures

4.8.1 RPI specific process data

LAM defines the type `struct _proc` to encapsulate all relevant information about an MPI process. Among others, there is a field called `p_rpi` for adding RPI specific process

data. Therefore, the type of this field lam_ssi_rpi_proc has to be defined by the RPI module. Figure 4.11 shows the one that is defined by the VIA RPI.

- cp_bmsg: This needs to be here for the unexpected message handling to save an incoming body of a message, if there is no matching request known on receiver side.
- cp_proc: Points up to the main _proc. This is an optimization within all RPI modules, where usually just a (struct lam_ssi_rpi_proc*) is passed around, because the most work is done with just this. But periodically, a few things from the upper-level _proc are needed.
- cp_pending_recvs: Queue for the pending receive requests on this _proc. If a message envelope arrives, this is used to search for a matching request. If none is found, the message is treated as unexpected.
- cp_pending_sends: Queue for the pending send requests on this _proc. It is necessary to find the right request and with it, the right callback function that has to be called If a VIA send operation has completed.
- cp_current_recv_req: Queue for requests that has been partially received. A req is queued up when the fist part of a short or long message (the envelope) has arrived.
- cp_vi_hand: This is the identification handle of the VI that is used for communication with the process that is represented by the _proc that contains this process data (p_rpi).
- cp_remote_nic_addr: The net address of the the remote NIC. It is not yet used exept for the initial connection. There is no need to save this at the moment but but we did because it will will be necessary when checkpoint/restart hooks are added to the RPI.

```
struct lam_ssi_rpi_proc {
  struct lam_ssi_rpi_cbuf_msg       *cp_bmsg;
  struct _proc                      *cp_proc;
  struct lam_ssi_rpi_via_reqlist    cp_pending_recvs;
  struct lam_ssi_rpi_via_reqlist    cp_pending_sends;
  struct lam_ssi_rpi_via_reqlist    cp_current_recv_req;
  VIP_VI_HANDLE                     cp_vi_hand;
  VIP_NET_ADDRESS                   *cp_remote_nic_addr;
  VIP_DESCRIPTOR                   *cp_send_desc,
  VIP_MEM_HANDLE                    cp_send_desc_hand,
  int                               cp_first_free_send_desc;
  int                               cp_free_send_desc_cnt;
  VIP_DESCRIPTOR                    *cp_recv_desc;
  VIP_MEM_HANDLE                    cp_recv_desc_hand;
  char                              *cp_rdma_area;
  VIP_MEM_HANDLE                    cp_rdma_area_hand;
  char                              *cp_remote_rdma_area;
  VIP_MEM_HANDLE                    cp_remote_rdma_area_hand;
  char                              *cp_read_pointer;
  char                              *cp_remote_write_pointer;
} lam_ssi_rpi_proc_t;
```

Figure 4.11: lam_ssi_rpi_proc: RPI specific process data (rpi_via_proc.h)

- `cp_send_desc`: An array of descriptors that may be used for sending data to the peer process.
- `cp_send_desc_hand`: Memory handle of the array of send descriptors.
- `cp_first_free_send_desc`: First send descriptor that is ready for posting it to the send queue.
- `cp_free_send_desc_cnt`: Number of free send descriptors.
- `cp_recv_desc`: Array of receive descriptors.
- `cp_recv_desc_hand`: The memory registration handle of the array of receive descriptors.
- `cp_rdma_area`: Pointer to an area within the local memory where the peer process may write data to.
- `cp_rdma_area_hand`: Memory registration handle of the RDMA area.
- `cp_remote_rdma_area`: Pointer to the beginning of the RDMA area of the peer. This is needed to reset the write pointer correctly, in case of reaching the end of the RDMA area.
- `cp_remote_rdma_area_hand`: Memory registration handle of the remote RDMA area. Whenever a short or tiny send to the peer shall be processed, this is needed to setup a send descriptor.
- `cp_read_pointer`: Points to the first bit of the next message that had arrived and has not been handled yet.
- `cp_remote_write_pointer`: Points to the first byte of the remote RDMA area that is free for overwriting.

4.8.2 RPI specific request data

A request is represented by an instance of type `struct _req`. Similar to the `struct _proc`, this record contains a field for adding RPI-specific data too. It is called `p_rpi`, and the name of its type is `lam_ssi_rpi_req`. See Figure 4.12.

```
typedef struct  lam_ssi_rpi_req {
  int                                cq_state;
  int                                cq_peer;
  struct lam_ssi_rpi_via_envl        *cq_envbuf;
  VIP_MEM_HANDLE                     cq_envhand;
  char                               *cq_packbuf;
  int                                cq_packsize;
  VIP_MEM_HANDLE                     cq_packbufhand;
  int                                cq_isadv;
  lam_ssi_rpi_via_send_progress_fn  cq_send_advance_fn;
  lam_ssi_rpi_via_recv_progress_fn  cq_recv_advance_fn;
  lam_ssi_rpi_via_send_callback_fn  cq_callback_fn;
} lam_ssi_rpi_req_t;
```

Figure 4.12: lam_ssi_rpi_req: RPI specific request data (rpi_via_req.h)

- `cq_type`: Kind of this request. Defined values are:
  - `C2CWRITE`: It is a send|isend|bsend|ssend.
  - `C2CREAD`: It is a receive.
  - `C2CSENDSELF`: It is a send (sender = receiver)
- `cq_peer`: Rank of the peer process.

- `cq_envbuf`: Pointer to pinned memory that is given back by `lam_ssi_rpi_via_dma_env_malloc`. It is used to store and send envelopes.
- `cq_envhand`. Pinned memory handle for the envelope buffer.
- `cq_packbuf`: A copy of the requests data buffer pointer `rq_packbuf`. Because VIA defines a maximum transfer size, we have to split larger messages into smaller packets. This pointer holds the address of the first not yet sent byte of the requests data. It is updated during the send process after each sent packet.
- `cq_packsize:`: Size of the still to sent data.
- `cq_packbufhand`: Memory handle of cq_packbuf. This keeps the handle for the user buffer of outgoing short and long messages.
- `cq_isadv`: Marks the request as already started and in the progression engine.
- `cq_send_advance_fn`: Holds a pointer to a function that continues the requests send activities depending on the state of the request and the protocol that has to be used. For example, sending the first package of a long message or sending an acknowledgment when data of a synchronous send have been received.
- `cq_recv_advance_fn`. Similar to `cq_send_advance_fn`, this variable holds a pointer to a function that continues the requests receive activities.
- `cq_callback_fn`: Pointer to the function that has to be called after the completion of a send descriptor.

4.8.3 The envelopes

It is not restricted by LAM what kind of data structure a RPI must use for saving envelope data and not even if there has to be an envelope. However, there is an prototype of an

envelope data structure named `lam_ssi_rpi_envl` that for the moment all included RPIs use, and is suggestive to use this for the VIA RPI because it bunches basic values for message identification such as size and tag. To allow the realization of all communication concepts of the VIA RPI, especially the rendezvous protocol, it is necessary to add some information. We followed the way all included RPIs go by using `lam_ssi_rpi_envl` as a member of an own envelope type `lam_ssi_rpi_via_envl` as shown in figure 4.13

```
typedef struct lam_ssi_rpi_via_envl {
  struct lam_ssi_rpi_envl   ve_env;
  char                      *ve_targetBuf;
  VIP_MEM_HANDLE            ve_targetBufHand;
  MPI_Request               ve_src_request;
} lam_ssi_rpi_via_envl_t;
```

Figure 4.13: lam_ssi_rpi_via_envl: RPI specific envelop data (rpi_via_envl.h)

- `ve_env`: Instance of the global envelope type that holds basic message identification values (`ce_len`, `ce_tag`, `ce_flags`, `ce_rank`, `ce_cid`, `ce_seq`).
- `*ve_targetBuf`: In order to make the communication peer known of the address of a requests user buffer (long protocol), this address is stored and transmitted within this field.
- `ve_targetBufHand`: The memory registration handle that belongs to `*ve_targetBuf`.
- `ve_src_request`: A process that initiates a data exchange stores the pointer of the request that is the reason for this communication in `ve_src_request`. In case the receiver has to send an acknowledgment, this pointer will be sent back to the sender to make sure that this one will be able to assign the incoming message to the request it belongs to.

4.8.4 Unexpected messages

As explained above (4.7.5), the RPI uses LAMs general interface for buffering unexpected messages. The corresponding data structure can be seen in figure 4.14. Figure 4.15 shows the additional values that the VIA RPI needs.

```
struct lam_ssi_rpi_cbuf_msg {
  struct _proc                *cm_proc;
  struct lam_ssi_rpi_envl     cm_env;
  struct lam_ssi_rpi_cbuf     *cm_extra;
  char                        *cm_buf;
  int                         cm_dont_delete;
  MPI_Request                 cm_req;
};
```

Figure 4.14: lam_ssi_rpi_cbuf_msg: Unexpected message data (rpisys.h)

- `cm_proc`: The source process (_proc) of the unexpected message.
- `cm_env`: The unexpected received envelope.
- `cm_extra`: RPI specific supplements.
- `cm_buf`: Additional buffer space, usually used for the body of a message.
- `cm_dont_delete`: This indicates that the cm_buf must not be freed when this cbuf_msg is freed.
- `MPI_Request cm_req`: If there is a send request specifying the send process to be the same as the receiver, the message will be stored as an unexpected message with this field set to the send request. When the receive request gets started, it will match this buffered message. As, with this field, the send request is known too, both requests can be marked done immediately.

```
typedef struct lam_ssi_rpi_cbuf {
  MPI_Request lsrc_src_request;
} lam_ssi_rpi_cbuf_t;
```

Figure 4.15: lam_ssi_rpi_cbuf: RPI specific unexpected message data (rpi_via_cbuf.h)

- `lsrc_src_request` The source request is currently the only piece of data that has to be saved from the initial envelope from the sender. For the RPI specific envelope there is already a field in `lam_ssi_rpi_cbuf_msg`.

5 Configuration, compilation and installation of the VIA RPI

In order to plug the VIA RPI into the LAM, the VIA RPI follows the requirements of a SSI module. This is realized by creating autoconf and automake files, which are detected and executed by the LAM runtime environment. Interventions into the LAM installation routines are not necessary. In these scripts the existence of required libraries is tested and initial values for the parameters, the user can consign to the RPI module on runtime, are set.

5.1 Requirements

The VIA RPI needs the following libraries installed on the system in order to build and configure.

- M-VIA library for driving the VIA devices
- pthread library for the thread support

5.2 Installation

The module can be involved into LAM by copying the VIA RPI directory into the `share/ssi/rpi` sub folder of the LAM installation. If a CVS snapshot is used auto-

tools are needed in order to create all necessary configure and Makefiles. They are called by the autogen.sh script in the LAM-MPI top level directory. For linking the libraries the libtool library ≥1.5 is required. After executing this script,the installation process of LAM can be continued with the configuration step. Every parameter for the VIA RPI module has to be passed to the global configure script. The specific parameters for the VIA RPI are the possible:

- `--with-rpi=via`
  sets the VIA RPI as the default RPI. On runtime LAM-MPI automatically chooses the VIA RPI as the default transmission module.

  mpirun parameter : -ssi rpi via

- `--with-rpi-via=<path>`
  Directory where the VIA software is installed. This directory must contain an include folder with vipl.h and an lib folder with libvipl.a inside.

- `--with-rpi-via-device=NAME`
  use NAME as the device name of the VIA board (default: /dev/via_eth0)

  mpirun parameter : `-ssi rpi_via_device <NAME>`

- `--with-rpi-via-tiny=BYTES`
  use BYTES as the size of the longest VIA tiny message (default: 16384)

  mpirun parameter : `-ssi rpi_via_tiny <BYTES>`

- `--with-rpi-via-short=BYTES`
  use BYTES as the size of the longest VIA short message (default: 32768)

  mpirun parameter : `-ssi rpi_via_short <BYTES>`

- `--with-rpi-via-area=BYTES`
  use BYTES as the size of the RDMA-able memory area for messages of one other process (default: 65536)

mpirun parameter : `-ssi rpi_via_rdesc <BYTES>`

- `--with-via-rdesc=COUNT`
  This is the count of pre-posted receive descriptors per connection. It defines concurrently the limit of of messages that can be received from a remote process at once. (default: 50)

  mpirun parameter : `-ssi rpi_via_rdescs <COUNT>`

- `--with-via-sdesc=COUNT`
  COUNT is the number of send descriptors per process that are consumed for sending messages per remote process(default: 10)

  mpirun parameter : ă`-ssi rpi_via_sdescs <COUNT>`

The last six parameters as well as the RPI that shall be used can be changed on runtime by passing parameters to `mpirun`. The less the count of send and receive descriptors are the more problable it is that the RPI chrashes. The reason are too few descriptors for a too high messages occurrence.

As usual the installation process can be finished by executing `make` and `make install`. For further details the LAM-MPI installation guide [9] is a very helpful source.
With the new version of LAM-MPI ($\geq$7.1) it is also possible to plug closed source modules of SSI kinds into an already existing environment. In order to use this capability LAM-MPI must be configured with the following options:

- `--with-modules`
- `--enable-shared`
- `--disable-static`

After that, desired precompiled SSI libraries can be copied into the lib folder of the directory that was specified with the `--prefix` option of the configure script and LAM-MPI will detect and include these modules dynamically.

6 Benchmarks

6.1 Comparable software

Standard MPICH MPICH is the most often used MPI software. For our tests we have worked with version 1.2.5. with TCP channel device.

LAM-MPI 7.1 with TCP RPI We used the latest version in the cvs tree. The TCP RPI was compiled with the default options given by LAM-MPI.

ParMa^2: LAM/MPI over M-VIA This solution was developed at the university of Parma and represents the M-VIA variant of LAM-MPI 6.3.2. This version is obsolete and unfortunately it was impossible to get this version running because of mistakes in the public source code. It was unable to be compiled. So we could not include it in our benchmark analysis.

MVICH : MPICH over M-VIA This is the only implementation of a VIA channel device for MPICH that works. It is the reference implementation of the M-VIA developers. It only runs with MPICH-1.2.2.3 and older versions.

6.2 Testbeds

The Testbed consists of PCs with the following configuration :

Cluster 1

- CPU : 2 AMD Athlon(tm) MP 1600+ 1,4 GHz 256 Kb cache size
- RAM : 512 MB
- Network cards :
  Intel Ethernet Pro 100 FE card (the VIA device)
  3Com Corporation 3c905C-TX FE card (service network)
- Operating system : Red Hat Linux 7.3
- Kernel: linux 2.4.18-27.7.xnfsroot
- Compiler : gcc 2.96
- M-VIA : 1.2 with VIA eepro100 driver

Cluster 2

- CPU : Intel PentiumIII, 800 MHz
- RAM : 512 MB
- Network cards :
  Level One FNC 0108TX (the VIA device)
  Level One FNC 0108TX (service network)
- Operating system : Red Hat Linux 7.3
- Kernel: kernel 2.4.18-10

- Compiler : gcc 2.96
- M-VIA : 1.2 with VIA tulip driver

6.3 Results

At this point it shall be shown whether the VIA RPI meets the requirements of an RPI for LAM-MPI and its performance. The first has been analyzed with the LAM-MPI test suite and is summarized in the first part of this chapter. In the further sections some benchmark results are given and evaluated. For measuring the performance of the MPI implementations we have chosen PALLAS Benchmark version 2.2
The section is subdivided into two important tests, Pallas supports. In order to help discerning the results, the alternative solutions are compared with our VIA RPI analysed detailed in each subsection.

6.3.1 Conformance

LAM-MPI brings a test suite for MPI implementations with it. It is created for finding bugs, and tests whether the requirements of the MPI standard are fulfilled. It is based on a modified version of the IBM test suite which was originally derived from an MPICH test suite. Tests that create more then one process on the same node must fail. This is not possible at the moment because our RPI is developed for the the multi RPI variant of LAM-MPI. The development of them is still in progress. At this new version several RPIs will run concurrently. This means for example : Communication on the same node is handled by an intranode communication RPI such as the shared memory RPI usysv and the internode communication is handled by the VIA RPI. All other tests have been passed. One of the disadvantages of the RPI that we noticed, during the run of the test suite, is the need for pinned memory. This resource is limited by the operating system and M-VIA.

It leads to memory shortage and failures if enough processes are involved or the transfer size arises. At the testbeds one process could only pin about 20 MB of memory. With 6 processes and an Alltoall with 4 MB packet size, the VIA RPI was overloaded and ended within a deadlock. The reason for that is that Alltoall creates many nonblocking request, which all try to pin memory simultaneously. The program had run out of pinnable memory because of many activated receive requests, yet before any send request could be stored. Two ideas for solving this problem are first, a rearrangement of none blocking requests and second, to allow unpinning of the memory of requests that are currently not being advanced.

6.3.2 PingPong

First we start to look on the results on cluster 1. Figure 6.1 gives an overview of the pingpong results of this cluster. It is obvious that the highest difference evolves from package sizes up to 512 bytes. The scaling of the VIA RPI curve is clean and without jumps and hooks. Our RPI also has the highest bandwidth exploitation too. Picture 6.2 gives a more detailed display on the range between 1 and 512 bytes. Over the whole range the VIA RPI has the fastest latency values that start with 74 us followed by MVICH with 77 us. It clearly shows that our solution can benefit from the short latency of the M-VIA device.

Figure 6.3 confirms, that the VIA RPI scales in the same manner as the pure M-VIA without any anomalies. If we follow the results in figure 6.1 the VIA RPI is always the fastest device, except for two parts.
The first is between 2048 to 4096 bytes. MVICH is faster in this range. The second interval is shown in figure 6.4. For message sizes between 16384 and 65536 LAM-MPI over TCP is quiet faster than over VIA. In order to understand this behavior, the results with the subtracted network latency are shown in figure 6.5

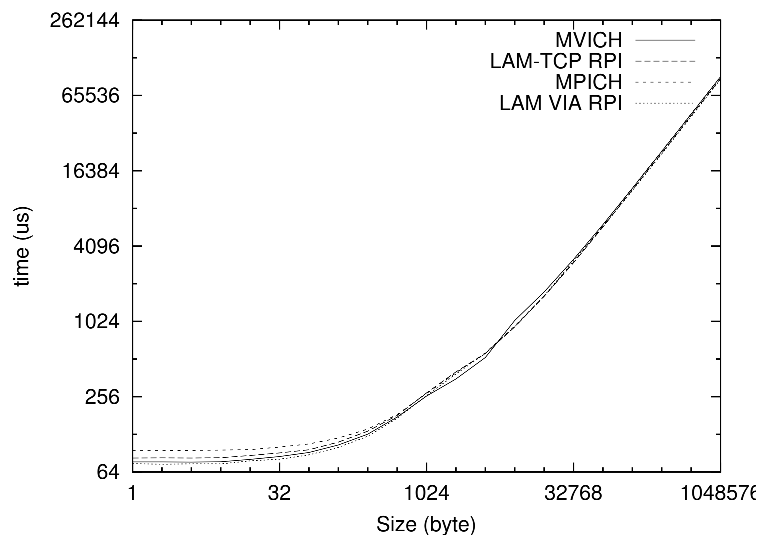

Figure 6.1: Pingpong on cluster 1

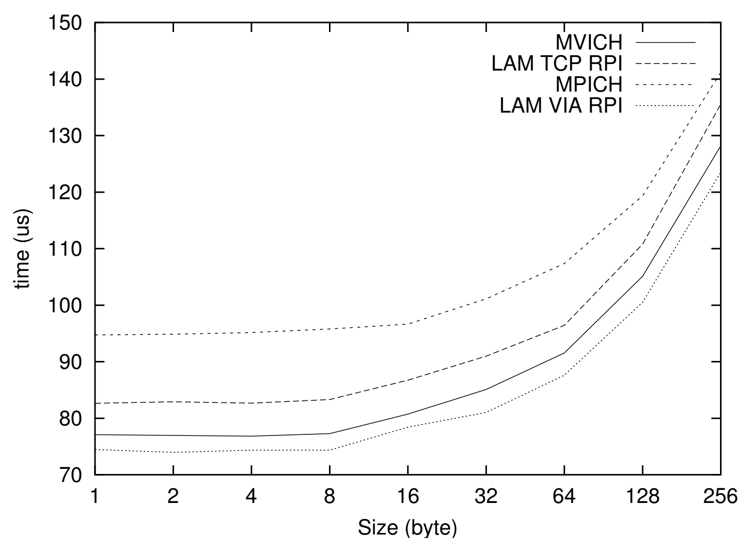

Figure 6.2: Pingpong latency details

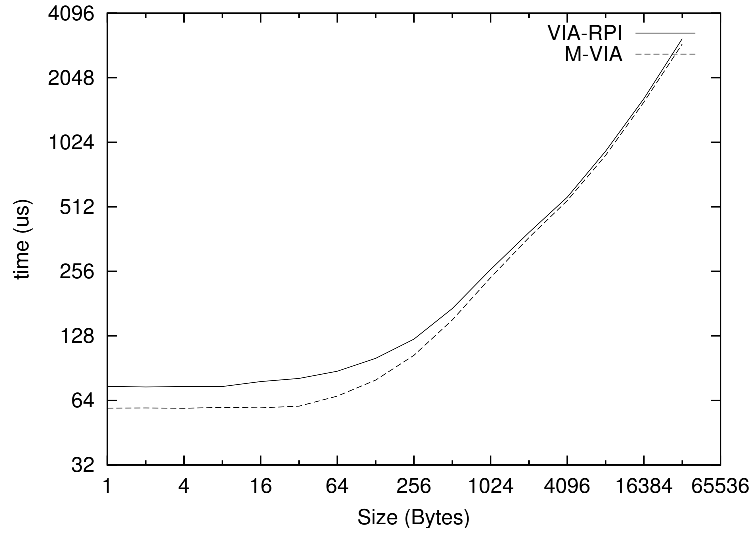

Figure 6.3: Comparison of the VIA RPI and the native M-VIA

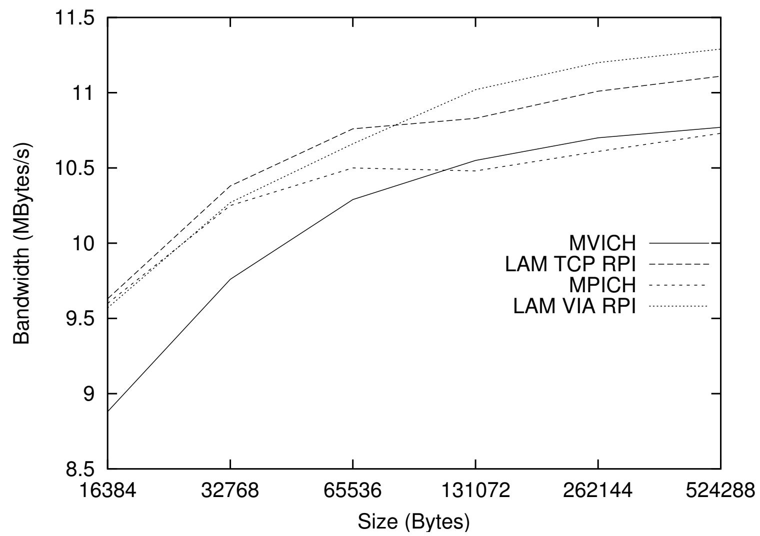

Figure 6.4: Pingpong bandwidth details

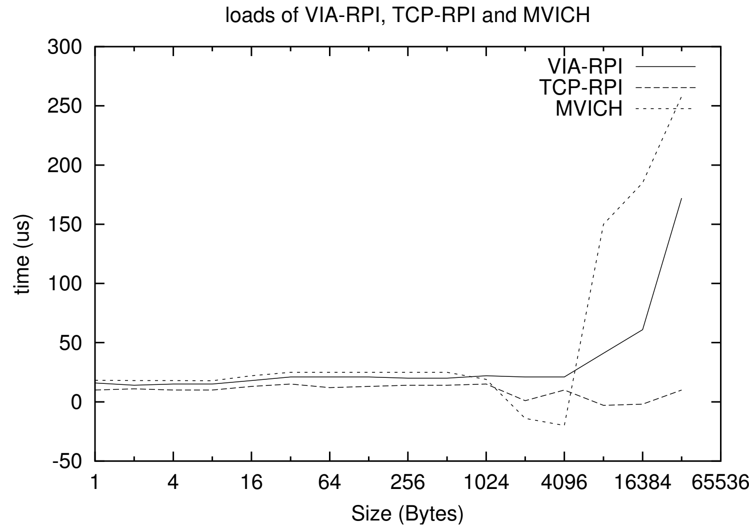

Figure 6.5: Load of the frameworks

The native latency of TCP was measured with the Netpipe benchmark and the latency of M-VIA with the PingPong test program brought along with M-VIA. In this figure it is shown that the overhead at these ranges of MVICH and the TCP RPI is lower than zero. This is impossible but measured. Because of our little knowledge of the internals of the other solutions, we actually can not explain these values. A possible explanation is that the TCP performance boost depends on the size of socket window. If this value is chosen cleverly, the data are transmitted faster for a designated message size.

In order to confirm our benchmarks we will examine on cluster 2 at figure 6.6. In this environment the result differs from the first. Obviously M-VIA makes a high performance jump. The TCP RPI is much slower than the VIA RPI. It is visible that MVICH is better for small packages up to 3500 bytes, but only on cluster 2. It can be explained with the lower cpu load of MVICH. This influences cluster 2 much more because of its three times slower CPU speed. For bigger packages the VIA RPI is the fastest because it benefits from the advantage of the pinning engine that is used for long and short messages as described in section 4.6.3.

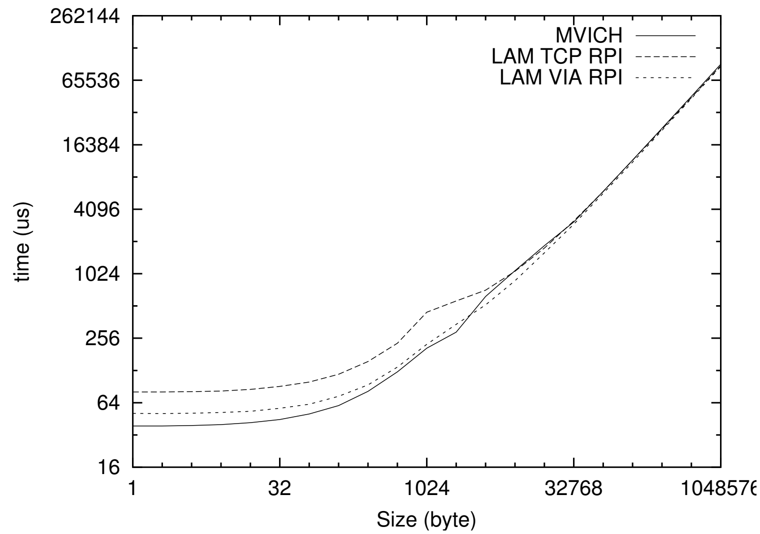

Figure 6.6: Pingpong on cluster 2

6.3.3 Broadcast

The behavior of performance values in case of collective operations is evaluated by the broadcast test of Pallas. Figure 6.7 clearly shows that the speed of the broadcast increases by using the VIA RPI. For short message sizes the latency for the TCP RPI is more than twice as big as compared to the VIA RPI. It is also obvious that MVICH is not suitable for collective operations. In fact, the TCP RPI is much faster than MVICH. It shows that MVICH has short latency if only two processes are involved. But if the count of communication rises up, MVICH slumps. Consequently the VIA RPI is the right choose for collective operations because of its strength of exhausting the bandwidth optimal.

Figure 6.8 gives us a more precise view on long messages.The VIA RPI can keep up with the other implementations also in this case, but the distance to MVICH becomes smaller.

The last figure (6.9) shows the results on cluster 2. Up to 65536 bytes this graph confirms the test on cluster 1. Beyond this value the speed of the VIA RPI bates. We found out, that the reason is the recurrent pinning and unpinning of memory because of its rar resources

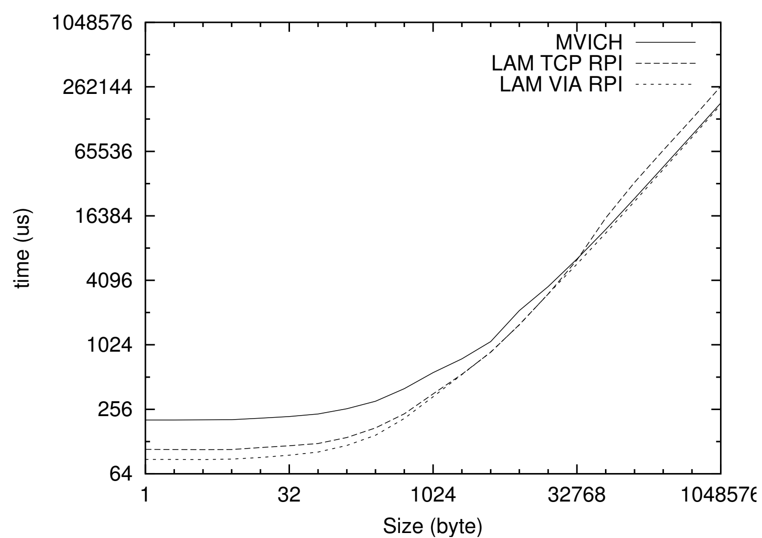

Figure 6.7: Broadcast with 4 nodes on cluster 1

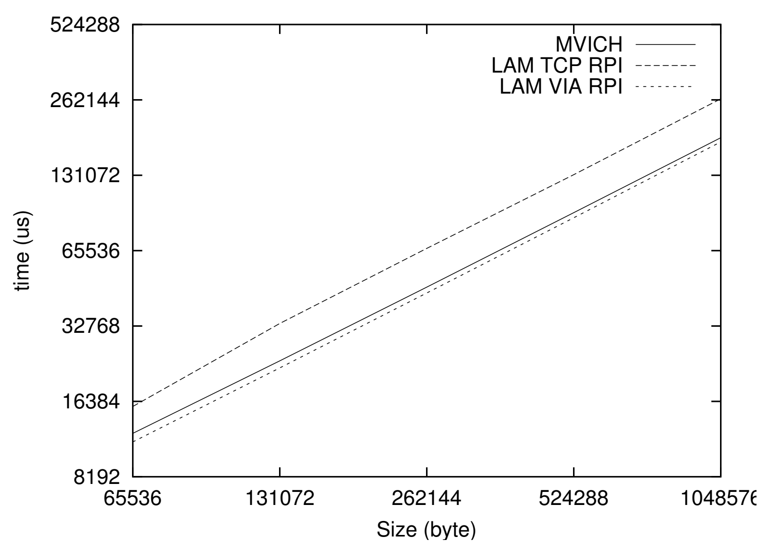

Figure 6.8: Broadcast details on cluster 1

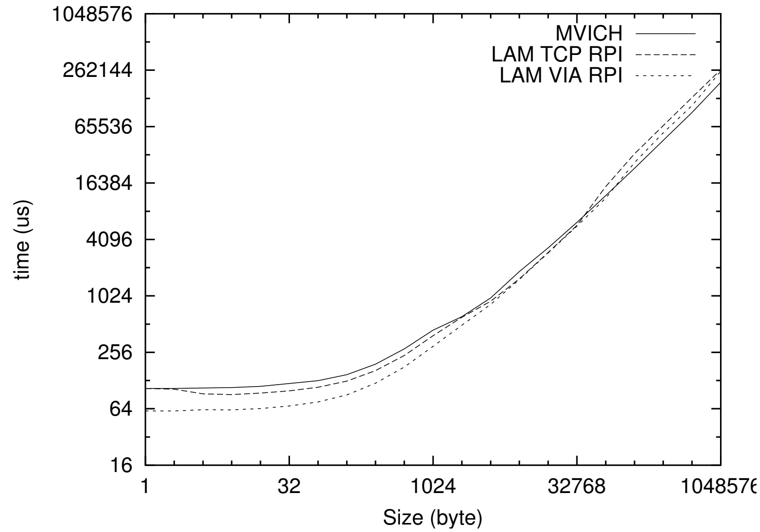

Figure 6.9: Broadcast with 4 nodes on cluster 2

of pinnable memory. If the already pinned memory such as the RDMA area (see 4.6.1)is minimized, the VIA RPI gets faster for long messages.

6.4 Conclusion

The goal was to develop a VIA RPI for LAM with less latency and higher bandwidth than the TCP RPI and MVICH. The benchmarks have exposed that the VIA RPI is an alternative to the TCP RPI - At least on clusters that are equipped with ethernet(100/1000) network cards M-VIA supports. It is nearly always faster for point to point as well as for collective operations. MVICH is faster for small messages but only for two processes. The more simultanous data transfers there are, the slower MVICH becomes. Therefore, all collective operations are less performant than with the new LAM RPI. All postulated conformance tests hav passed and the module was able to be installed successfully. It is ready for use but enhancements are still possible.

Bibliography

[1] M. Bertozzi, M. Panella, M. Reggiani *Design of a VIA based communication protocol for LAM/MPI suite* Parma University, 2001

[2] Intel Corporation *VIDF Virtual Interface (VI) Architecture Developer's Guide Revision 1.1* Intel Corporation, September 2000

[3] Intel Corporation *Virtual Interface Architecture Specification Version 1.0* Intel Corporation, 1997

[4] M-VIA: A High Performance Modular VIA for Linux http://www.nersc.gov/research/FTG/via/

[5] Libdict Dictionary Library, http://home.earthlink.net/∼smela1/libdict.html, 2003

[6] Jeffrey M.Squyres and Brian Barrett and Andrew Lumsdaine, *The System Services Interface (SSI) to LAM/MPI* Indiana University, Computer Science Department, 2003

[7] Jeffrey M.Squyres and Andrew Lumsdaine, *A Component Architecture for LAM/MPI* Indiana University, Computer Science Department, 2003

[8] Jeffrey M.Squyres and Brian Barrett and Andrew Lumsdaine, *Request Progression Interface (RPI) Modules for LAM/MPI* Indiana University, Computer Science Department, 2003

[9] The LAM/MPI Team Open Systems Lab, LAM/MPI Installation Guide Version 7.0.2, Indiana University, Computer Science Department, 2003

[10] The LAM/MPI Team Open Systems Lab, LAM/MPI User Guide Version 7.0.2, Indiana University, Computer Science Department, 2003

[11] Pallas GmbH, *Pallas MPI Benchmarks - PMB, Part MPI-1* Pallas GmbH, March 2000

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die in der Tabelle mir zugeschriebenen Kapitel der vorliegenden Arbeit, selbstständig angefertigt habe. Ich habe die Arbeit nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 29. Januar 2004

Ralph Engler

Tobias Wenzel

von Ralph Engler bearbeitete Kapitel
1, 2, 3, 4.2.1, 4.4.3,
4.6 intro, 4.6.3, 5, 6

von Tobias Wenzel bearbeitete Kapitel
4.1, 4.2.2, 4.3, 4.4.1, 4.4.2,
4.5, 4.6.1, 4.6.2, 4.7, 4.8